NASA Technical Paper 3491

Design Tool for Multiprocessor Scheduling and Evaluation of Iterative Dataflow Algorithms

Robert L. Jones III
Langley Research Center • Hampton, Virginia

National Aeronautics and Space Administration
Langley Research Center • Hampton, Virginia 23681-0001

April 1995

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Acknowledgments

This paper has benefited from numerous discussions with Sukhamoy Som of Lockheed Engineering & Sciences Company and Paul Hayes of the Langley Research Center. Rodrigo Obando of Old Dominion University and Asa Andrews of CTA, Inc., provided invaluable technical discussions during the software implementation of the Design Tool. Asa Andrews developed the Graph-Entry Tool.

Available electronically at the following URL address: http://techreports.larc.nasa.gov/ltrs/ltrs.html

Printed copies available from the following:

NASA Center for AeroSpace Information
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

Contents

Tables

Figures

Nomenclature

AMOS	ATAMM multicomputer operating system
ATAMM	algorithm to architecture mapping model
C_i	sum of node latencies in *i*th circuit
CMG	computational marked graph
$D_f(T)$	relative data set associated with finish of task T
D_i	total number of tokens within *i*th circuit
$D_s(T)$	relative data set associated with start of task T
DFG	dataflow graph
DSP	digital signal processing
d	number of initial tokens on edge or within path
EF	earliest finish time
ES	earliest start time
$F(T)$	TBO-relative finish time of task T
GVSC	generic VHSIC spaceborne computer
L_i	*i*th element in $\mathcal{L}$; latency of *i*th task
$\mathcal{L}$	set of task latencies
LF	latest finish time
$\mathcal{M}_o$	initial marking of graph
MDFG	modified dataflow graph
N_i	*i*th node in DFG
OE	output empty; number of initially empty output queue slots
OF	output full; number of initially full output queue slots
$\mathcal{P}$	maximum data set number
PI	parallel-interface bus
R	total number of required processors
S	speedup
$S(T)$	TBO-relative start time of task T
SGP	single graph play
$\text{SGP}_{s\text{-}s}$	steady-state single graph play
$\text{SGP}_{t\text{-}s}$	transient-state single graph play
SRE	single resource envelope
T_i	*i*th task in $\mathcal{T}$
T_o	maximum time per token ratio for all graph circuits
t	time
$\mathcal{T}$	set of tasks
TBI	time between inputs
TBIO	time between input and output
TBIO_{lb}	lower bound time between input and output

TBO	time between outputs
TBO_{lb}	lower bound time between outputs
TCE	total computing effort
TGP	total graph play
TRE	total resource envelope
U	utilization
VHSIC	very-high-speed integrated circuit
Δ	EF – LF for a given task
ω	schedule length
$\prec$	partial ordering of tasks

Abstract

A graph-theoretic design process and software tool is defined for selecting a multiprocessor scheduling solution for a class of computational problems. The problems of interest are those that can be described with a dataflow graph and are intended to be executed repetitively on a set of identical processors. Typical applications include signal processing and control law problems. Graph-search algorithms and analysis techniques are introduced and shown to effectively determine performance bounds, scheduling constraints, and resource requirements. The software tool applies the design process to a given problem and includes performance optimization through the inclusion of additional precedence constraints among the schedulable tasks.

1. Introduction

This paper describes methods capable of determining and evaluating the steady-state behavior of a class of computational problems for iterative parallel execution on multiple processors. The computational problems must be capable of being described by a directed graph. When the directed graph is a result of inherent data dependencies within the problem, the directed graph is often referred to as a "dataflow graph." Dataflow graphs, generalized models of computation, have received increased attention for use in modeling parallelism inherent in computational problems (refs. 1 through 3). This attention can be attributed not only to the ease at which dataflow graphs can model parallelism but also in their amenability to direct interpretation of program flow and behavior (ref. 4).

In this paper, graph nodes represent schedulable tasks and graph edges represent the data dependencies between the tasks. Because the data dependencies imply a precedence relationship, the tasks make up a partial-order set; that is, some tasks must execute in a particular order, whereas other tasks may execute independent of other tasks. When a computational problem or algorithm can be described with a dataflow graph, the inherent parallelism present in the algorithm can be readily observed and exploited. The modeling methods presented in this paper are applicable to a class of dataflow graphs where the time to execute tasks is assumed constant from iteration to iteration when executed on a set of identical processors. Also, the dataflow graph is assumed to be data independent; that is, any decisions present within the computational problem are contained within the graph nodes rather than described at the graph level. The dataflow graph provides both a graphical and mathematical model capable of determining run-time behavior and resource requirements at compile time. In particular, dataflow graph analysis is shown to be able to determine the exploitable parallelism, theoretical performance bounds, speedup, and resource requirements of the system. Because the graph edges imply data storage, the resource requirement specifies the minimum amount of memory needed for data buffers as well as the processor requirements. Obtaining this information is useful in allowing a user to match the resource requirements with resource availability. In addition, the nonpreemptive scheduling and synchronization of the tasks that are sufficient to obtain the theoretic performance are specified by the dataflow graph. This property allows the user to direct the run-time execution according to the dataflow firing rules (i.e., when tasks are enabled for execution) so that the run-time effort is reduced to simply allocating an idle processor to an enabled task (refs. 5 and 6). When resource availability is not sufficient to achieve optimum performance, a technique of optimizing the dataflow graph with artificial data dependencies, called control edges, is discussed.

Predicting the computing performance, resource requirements, and processor utilization connected with the execution of a dataflow graph requires the determination of steady-state behavior. Dataflow graph analysis algorithms and rules are defined in this paper for determining the scheduling constraints, that is, earliest execution times and mobility, for all tasks under steady-state conditions. It is also shown that certain initial conditions represented by initial data in a dataflow graph may result in a transient-state execution different from the steady-state execution. The analysis algorithms are shown to detect such transient conditions. The method for determining periodic steady-state behavior is based on first describing the execution of data associated with a single computational iteration, referred to as a "data set." Second, the transient state is distinguished from the steady state if necessary when initial data are present. Finally, the periodic execution for multiple iterations is determined from the steady-state single iteration description.

For the mathematical models presented, an efficient software tool which applies the models is desirable for solving problems in a timely manner. A software tool developed for design and analysis is presented. The software program, referred hereafter as the "Design Tool," provides automatic and interactive analysis capabilities applicable to the design of a multiprocessing solution. The development of the Design Tool was motivated by a need to adapt multiprocessing computations to emerging very-high-speed integrated circuit (VHSIC) space-qualified hardware for aerospace applications. In addition to the Design Tool, a multiprocessing operating system based on a directed-graph approach called the ATAMM multicomputer operating system (AMOS) was developed. AMOS executes the rules of the algorithm to architecture mapping model (ATAMM) and has been successfully demonstrated on a generic VHSIC space-borne computer (GVSC) consisting of four processors loosely coupled on a parallel-interface (PI) bus (refs. 5 and 6). The Design Tool was developed not only for the AMOS/GVSC application-development environment presented in references 5 and 7 but for other potential dataflow applications. For example, the design procedures based on ATAMM solve signal processing problems addressed by Parhi and Messerschmitt in reference 3. (See ref. 8.) Information provided by the Design Tool could also be used as scheduling constraints as done in reference 9 to aid other scheduling algorithms.

The modeling of a computational problem with a dataflow graph and analysis diagrams is discussed in section 2. A forward-search algorithm is defined and is shown to determine the earliest execution times for all tasks. Section 3 discusses a modification to the dataflow graph described in section 2, which lends itself to the modeling of initial conditions. In addition, a backward-search algorithm is defined and shown to determine the mobility of the tasks and transient conditions which affect the steady-state behavior. The performance metrics and resource requirements procedures implemented in the Design Tool are described in section 4. The memory requirements of data shared among tasks, as described by a directed graph, is shown to be bounded. Rules for determining the minimum memory requirements for buffering-shared data are defined. The Design Tool displays and features are presented in section 5 where the performance results are compared with the theoretical results derived in the previous sections. Section 5 also presents execution time results regarding the Design Tool implementation of the algorithms presented in sections 2 and 3. Applications and future research are summarized in section 6.

2. Dataflow Graphs and Scheduling Diagrams

A generalized description of a multiprocessing problem and how it can be modeled by a directed graph is presented in this section. Such formalism is useful in defining the graph analysis algorithms and rules which determine scheduling constaints. A computational problem (job) can often be decomposed into a set of tasks to be scheduled for execution (ref. 10). If the set of tasks are not independent of one another, a precedence relationship is imposed on the tasks in order to obtain correct computational results. A task system can be represented formally as a 4-tuple ($\mathcal{T}$, $\prec$, $\mathcal{L}$, $\mathcal{M}_o$) where

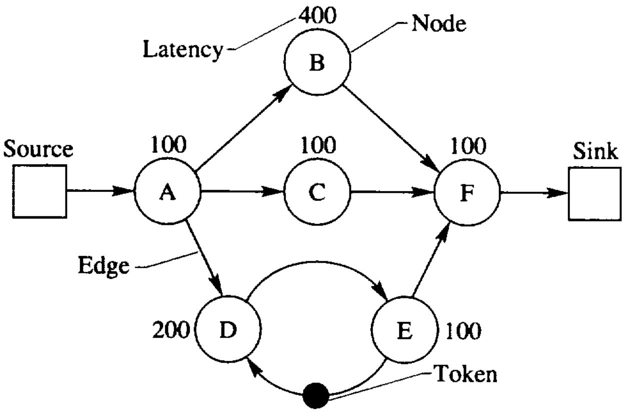


Figure 1. Dataflow graph.

$\mathcal{T}$ set of n tasks to be executed, $\{T_1, T_2, T_3, ..., T_n\}$

$\prec$ precedence relationship on $\mathcal{T}$ such that $T_i \prec T_j$ signifies that T_j cannot execute until completion of T_i

$\mathcal{L}$ nonempty, strictly positive set of run-time latencies such that task T_i takes L_i amount of time to execute, $\{L_1, L_2, L_3, ..., L_n\}$

$\mathcal{M}_o$ initial state of system, as indicated by presence of initial data

Such task systems can be described by a directed graph where nodes (vertices) represent the tasks and edges (arcs) describe the precedence relationship between the tasks. When the precedence constraints given by $\prec$ are a result of the dataflow between the tasks, the directed graph is referred to as a "dataflow graph (DFG)" as shown in figure 1. Special transitions called sources and sinks are also provided to model the input and output data streams of the task system. The presence of data is indicated within the DFG by the placement of tokens. The DFG is initially in the state indicated by the marking $\mathcal{M}_o$. The graph moves through other markings as a result of a sequence of node firings (executions); that is, when a token is available on every input edge of a node and sufficient resources are available for the execution of the task represented by the node, the node fires. When the node associated with task T_i fires, it consumes one token from each of its input edges, delays an amount of time equal to L_i, and then deposits one token on each of its output edges. Sources and sinks have special firing rules; sources are unconditionally enabled for firing, and sinks consume tokens but do not produce any. By analyzing the DFG in terms of its critical path, critical circuit, dataflow schedule, and the token bounds within the graph, the performance characteristics and resource requirements can be determined a priori. The Design Tool depends on this dataflow representation of a task system, and the graph-theoretic performance metrics presented herein.

The graph execution for a single iteration, unlimited resources assumed, can be portrayed with a Gantt chart where horizontal bars are used to indicate when tasks may be scheduled for execution. Such a chart is referred to hereafter as a "single graph play (SGP) diagram," which is shown in figure 2 for the DFG of figure 1. The SGP can be constructed by calculating the earliest start (ES) times for all tasks. The ES times can be calculated by envisioning the migration of a single data set through the graph. Since the condition for a node to fire (begin execution) is having a token present on all its inputs, the ES time for a given task is equal to the longest path latency (starting from the source) for all paths leading to its inputs. The longest input path latency would indicate the time at which all input tokens would be present for execution. The amount of time required for all nodes of a graph to execute a single data set or graph iteration is referred to as the schedule length, denoted as ω. For generality, the task latencies shown in figure 1 are given in `clock units`, and therefore the schedule length is shown in figure 2 to be equal to 600 clock units.

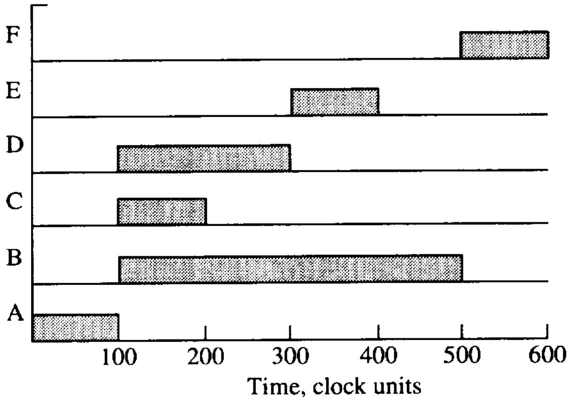
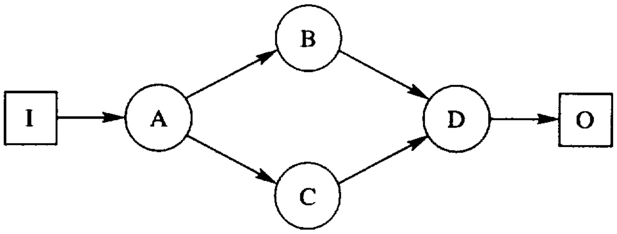


Figure 2. Single graph play diagram. ω = 600 clock units.

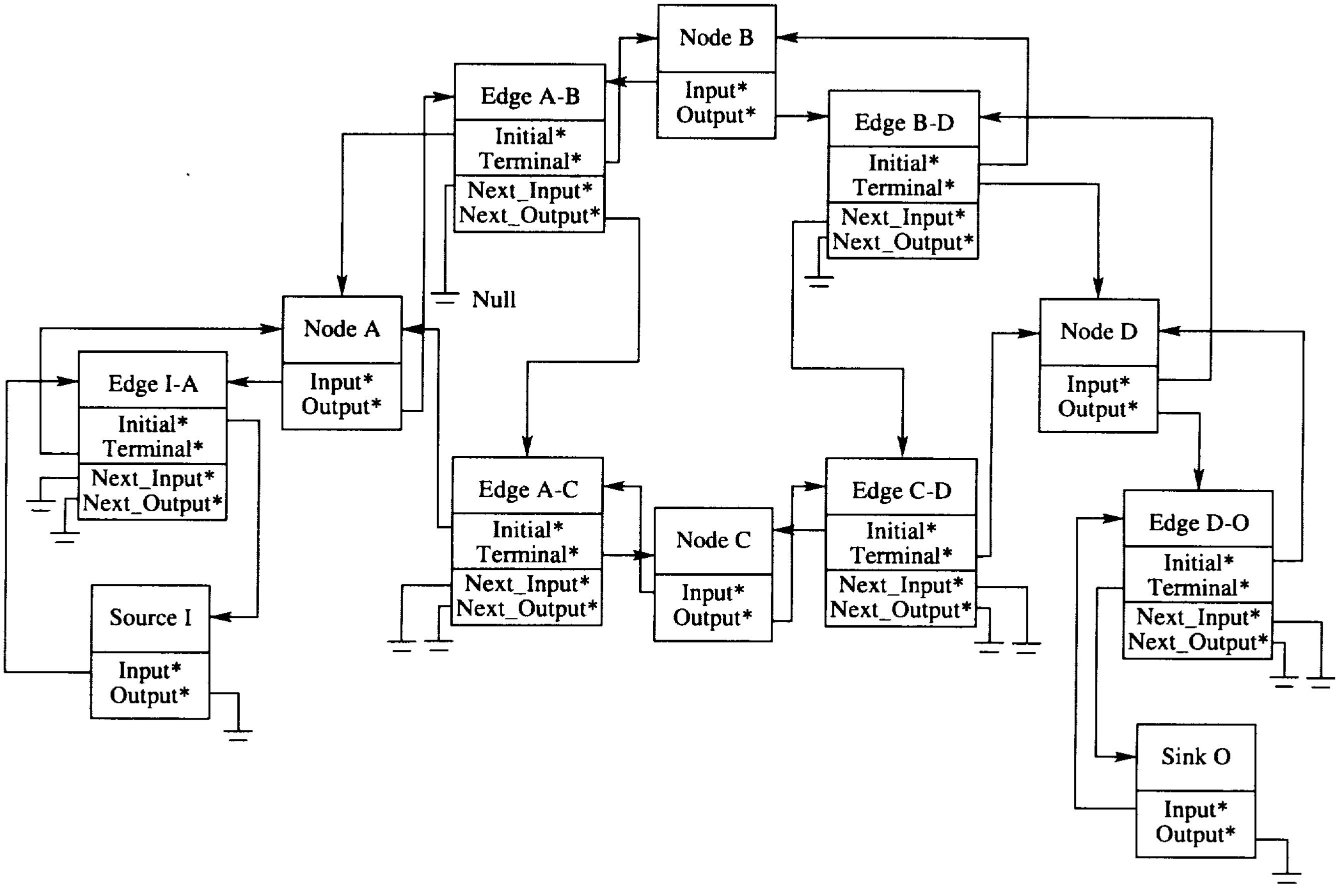
The two algorithms, defined in this paper, that implement a forward and backward search of the directed graph and other analyses are based on a linked-list representation of the graph. In this way, pointers can be used for efficient progression through the graph from any given starting point. An example illustrating the connections between node objects and edge objects is shown in figure 3. The object address pointers are denoted by asterisks. A node object points to just one `input` and one `output`. All other input and outputs are connected to the node by the `next input` and `next output` pointers. A `null` pointer indicates that no other input or output exists.

Given a linked-list graph representation as shown in figure 3, the following forward-search algorithm determines the earliest start times for all nodes (tasks). The algorithm employs the depth-first searching method where the graph is penetrated as deeply as possible from a given source before fanning out to other nodes. For each node encountered in the search, the algorithm calls the procedure `SearchFwd` recursively for each output edge associated with the node. The recursive nature of the algorithm allows a depth-first search of the graph to be done while implicitly retaining the next edge (starting point for the next path to traverse when fanning out) and accumulated path latency on the memory stack. The arguments passed into `SearchFwd` are an address pointer (`edge`) to an edge structure (fig. 3) and the current path latency (`path_latency`) up to the `edge`. Also, let `node` specify a pointer to a node structure. An `edge` will point to a `next_output` if present, and will be null if no other output edges for the current `node` exist. The `ES Algorithm` is stated as follows:

```
A. Initialize earliest start times for all nodes to
   zero

B. Execute procedure SearchFwd (source.
   output, 0) for every source in graph by start-
   ing with first output edge of source; path
   latency, the second parameter, initially set to
   zero

SearchFwd(edge, path_latency)

    1. If edge.next_output is not null,
       call SearchFwd (edge.next_out-
       put, path_latency).

    2. Get the node that uses this edge for
       input by setting node equal to
       edge.terminal_node.

    3. Determine the earliest start of node,
       ES (node), such that ES (node) = max
       [ES (node), path_latency].

    4. Increase path_latency by the node
       latency, $L_{node}$.

    5. Set edge equal to the first output edge of
       node, edge = node.output.

    6. If a sink has been reached (edge = null),
       return from this procedure; else repeat
       Step 1.
```



(a) Example graph.



(b) Linked-list representation.

Figure 3. Linked-list storage of dataflow graph.

The ES Algorithm execution time is graph dependent and is bounded by

$$\text{Bound} = \sum_{\text{Over all paths in DFG}} N_i \qquad (1)$$

where N_i is the number of nodes in a given path. Because the number of paths in a given graph with at most N nodes is bounded by N^2, the expression (eq. (1)) has a worst-case bound of N^3. Therefore, the ES Algorithm has a polynomial-time complexity of the order of N^3, or $O(N^3)$.

The elapsed time between the production of an input token by the source and the consumption of the corresponding output token by the sink is defined as the time between input and output (TBIO). When initial tokens are not present, ω will be equal to TBIO, otherwise ω may be greater than TBIO. As discussed later, the SGP determined by the ES analysis given by the ES Algorithm when initial tokens in the forward dataflow direction are present may not be representative of the steady-state behavior, $\text{SGP}_{s\text{-}s}$, at run time but instead portrays a transient state, $\text{SGP}_{t\text{-}s}$. Refinements to the computed earliest start times may be required to obtain the $\text{SGP}_{s\text{-}s}$. A method for determining these refinements is included in the next section.

Of particular interest are the cases when the algorithm modeled by the DFG is executed repetitively for different data sets. The iteration period and, thus, throughput is characterized by the metric TBO (time between outputs) where TBO is defined as the time between consecutive consumptions of output tokens by a sink. It can be shown that because of the consistency property of dataflow graphs, all tasks execute with period TBO (refs. 11 and 12). This implies that if input data are injected into the graph with period TBI (time between inputs) then output data will be generated at the graph sink with period TBO equal to TBI.

The periodic graph execution for multiple iterations can be portrayed in another Gantt chart referred to as a "total graph play (TGP) diagram." The TGP diagram shows the execution over a single iteration period of TBO. Like the single graph play diagram, the total graph play diagram represents task executions with horizontal bars. The TGP can be constructed from the SGP by dividing the SGP into segments of width TBO starting from the left of the diagram. The resulting SGP from the previous example for an arbitrarily selected TBO period of 333 clock units is shown in figure 4. Each segment is representative of the execution associated with a particular data set when the graph is executed periodically.

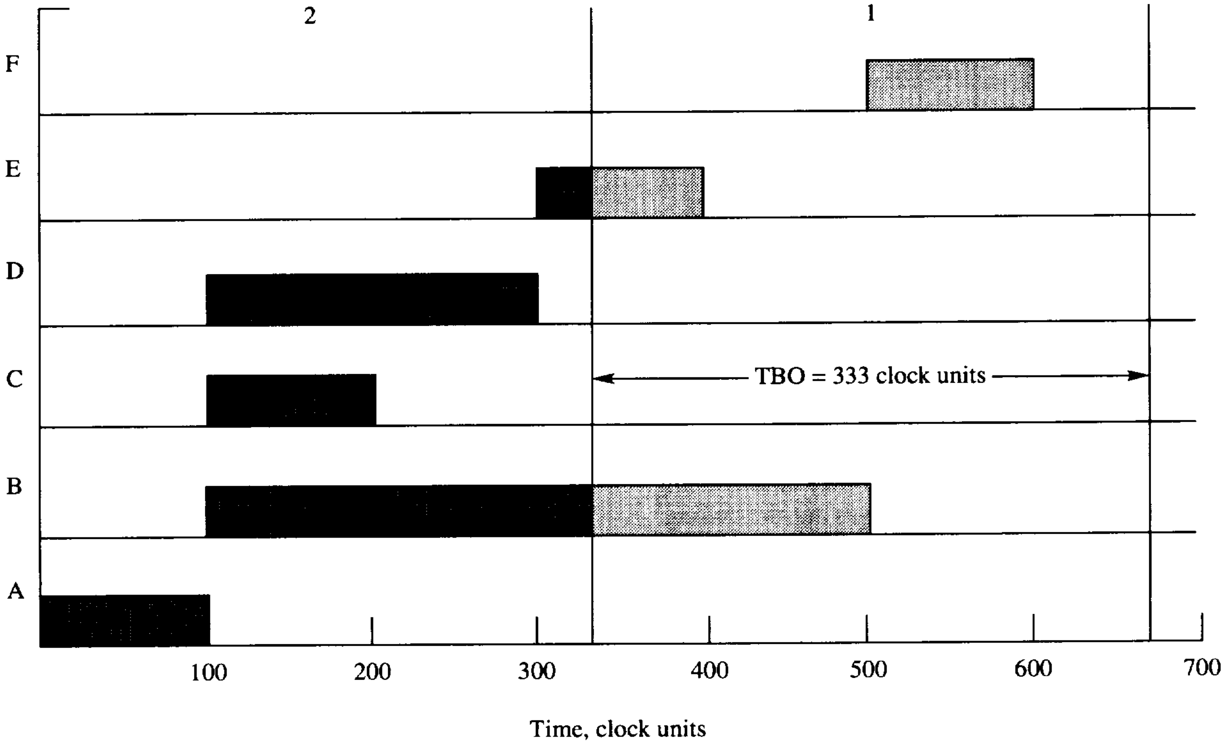


Figure 4. Segmented single graph play diagram.

Consequently, these segments are assigned relative data set numbers, 1 to $\mathcal{P}$, from right to left. Overlapping these segments portrays the graph execution for multiple data sets within a TBO period as shown in figure 5. Note that the relative data set numbers assigned to the task bars within the TGP of figure 5 correspond to the numbered SGP segments of figure 4. The fact that within a TBO period, every task will execute exactly once is obvious from the nature of how the TGP is constructed by overlapping TBO-width segments from the SGP. The total computing effort (TCE) within a TBO interval from SGP segments would therefore equal the sum of all task latencies within the latency set $\mathcal{L}$.

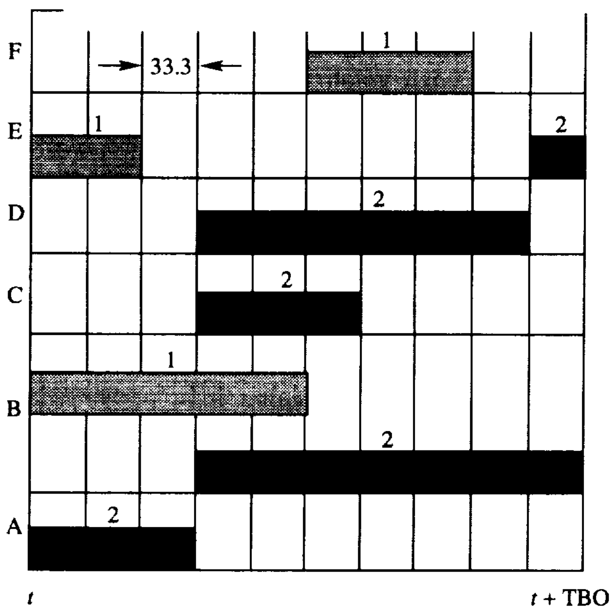


Figure 5. Total graph play diagram. TBO = 333 clock units.

Constructing the TGP by overlapping SGP segments is equivalent to mapping the ES times (relative to the SGP) to a time interval of width TBO by using the mapping function ES modulo TBO. The number of SGP segments is equal to the maximum number of data sets simultaneously present in the graph at steady state and indicates the level of pipeline concurrency that is being exploited. This metric is given by applying the ceiling[1] function to the ratio of the schedule length ω to TBO as shown in the following equation:

$$\mathcal{P} = \left\lceil \frac{\omega}{\mathrm{TBO}} \right\rceil \qquad (2)$$

By numbering the SGP segments 1 to $\mathcal{P}$ from right to left, a relative data set numbered $\mathcal{D}$ will refer to a data set injected into the graph 1 TBO interval after a data set numbered $\mathcal{D} - 1$. Overlapped bars for a given task indicate that the task has multiple instantiations as for task B. That is, the task is executed on different processors simultaneously for different data sets. Allowing multiple task instantiations is a key mechanism for increasing speedup.

The inherent nature of dataflow graphs is to accept data as quickly as the graph and available resources (processors and memory) allow. When this occurs, the graph becomes congested with tokens waiting on edges for processing because of the finite resources available, without resulting in an increase in throughput above the graph-imposed upper bound (refs. 2 and 13). When tokens wait on the critical path for execution, however, an increase in TBIO above the lower bound occurs. This increase in TBIO can be undesirable for many real-time applications. It is therefore necessary to constrain the parallelism that can be exploited in order to prevent resource saturation. Constraining the parallelism in dataflow graphs can be controlled by limiting the input injection rate to the graph. Adding a delay loop around the source makes the source no longer unconditionally enabled (ref. 5). It is important to determine the appropriate lower bound on TBO for a given graph and number of resources. Determination of the lower bound on TBO is deferred to section 4.

3. Dataflow Graph Analysis

In the absence of initial tokens within the graph, a latest finish (LF) time analysis would be similar to the depth-first searching method used to calculate the earliest start times, only in the reverse direction. That is, searching backward from all sinks, the latest time each task associated with an encountered node must complete in order to prevent an increase in the TBIO given by the ES time analysis can be determined. The latest finish time for a given task is equal to TBIO (for a given sink) less the maximum path latency to the associated node output from all possible paths leading backwards from the sink. The combination of earliest start and latest finish times provide the means to calculate the float or slack time that might be present for each task. Slack time indicates the maximum delay in task completion that can be tolerated without delaying the start times of successor tasks which result in an increase in TBIO. Slack time for a task is given by

$$\text{Slack time} = \mathrm{LF}(T_i) - \mathrm{ES}(T_i) - L_i \qquad (3)$$

with latency L.

[1] The ceiling of a real number x, denoted as $\lceil x \rceil$, is equal to the smallest integer greater than x.

When initial tokens are present within the graph, the ES and LF analysis presented here must be modified slightly. The method for determining the steady-state behavior of a dataflow graph when initial tokens are present is based on a simple extension to the earliest start time analysis described in the previous section and a latest finish time analysis to be discussed here. It will be shown in later examples that initial tokens within the DFG not only affect the calculations of ES and LF times but may also be associated with recurrence loops (resulting in graph circuits), which tend to complicate the graph search process. Modifications to the dataflow graph, which simplify the analysis, are defined here and can be shown to result in an equivalent model of the original graph. This modified dataflow graph is referred hereafter as the MDFG.

The MDFG can be constructed by letting all edges with one or more initial tokens undergo the transformation shown in figure 6 where such edges are terminated with "virtual" sinks. Each virtual sink is labeled with the identifier of the node that consumes tokens from the original edge. In the cases where all input edges of a node have initial tokens, a virtual source for each such node is added so that the node is not left dangling without an input edge. The addition of these virtual sources maintains compatibility with the `ES Algorithm`. The resulting MDFG of the dataflow graph in figure 1 is shown in figure 7.

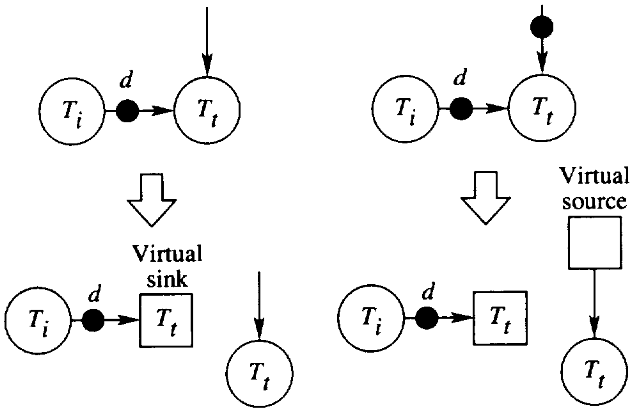


Figure 6. Constructing the modified dataflow graph.

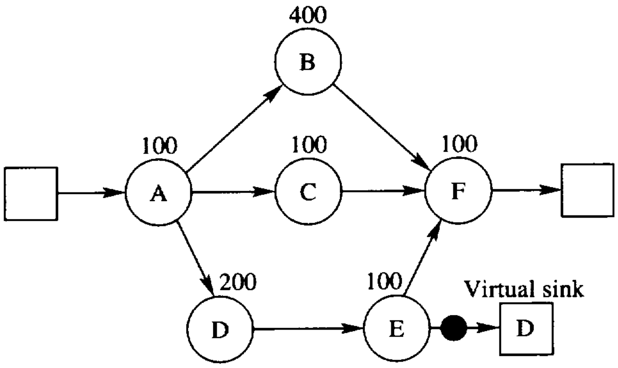


Figure 7. The modified dataflow graph equivalent of figure 1.

The MDFG can now model the more complex problem containing initial tokens but in a simpler, linear (source to sink) fashion. Now, the same ES analysis from all sources to sinks can be conducted as before. However, in order to ensure that the new MDFG is equivalent to the original dataflow graph, an additional time constraint must be imposed on the graph at these virtual sinks. Referring to figure 6, the time constraint is defined as follows:

$$\mathrm{LF}(T_i) = \mathrm{ES}(T_t) + d(\mathrm{TBO}) \tag{4}$$

where LF(T_i) represents the LF time of T_i due to the initial tokens, ES(T_t) represents the ES time of T_t, and d is the number of initial tokens on the $T_i \prec T_t$ edge. Stated in words, equation (4) determines the latest finish time of task T_i which returns a token on the edge initialized with d tokens such that the firing of task T_t will not be delayed. The ES(T_t) is determined by the `ES Algorithm` starting from all MDFG sources. If equation (4) results in a LF time *less* than the earliest finish (EF) time of T_i, a time constraint has been violated. Since a task cannot complete execution sooner than its earliest finish time (as determined from the ES analysis), a transient condition has been detected. For the first iteration, the graph will execute according to the $\mathrm{SGP}_{t\text{-}s}$ as defined by the `ES Algorithm`. However, since the next data set will arrive 1 TBO interval later, an additional time constraint will be imposed if initial tokens exist in the graph. The node T_t with d initial input tokens has the potential (depending on other input dependencies) of repeated firings until all d tokens are consumed. With each node firing with period TBO, the elapsed time to consume d tokens is the product of d and TBO. The predecessor node T_i must return a token within d(TBO) time relative to the ES so that the next firing of T_t is not delayed. Therefore, in order for node T_i to generate its first token in this timely manner which maintains the task schedule defined by the first iteration $\mathrm{SGP}_{t\text{-}s}$, it must do so by the time determined by equation (4). Otherwise, the firing of node T_t will be delayed, resulting in $\mathrm{SGP}_{s\text{-}s} \neq \mathrm{SGP}_{t\text{-}s}$.

Now that it has been shown that timing conflicts determined by equation (4) indicate the presence of a transient state, $\mathrm{SGP}_{t\text{-}s} \neq \mathrm{SGP}_{s\text{-}s}$, a method is needed to translate the $\mathrm{SGP}_{t\text{-}s}$ to the $\mathrm{SGP}_{s\text{-}s}$. By adjusting the earliest start times of the nodes affected by this delay, the steady-state behavior when initial tokens are present can be determined. When equation (4) indicates a timing conflict, determine the time difference between the result of equation (4), LF(T_i), and the earliest finish of the T_i, EF(T_i) = ES(T_i) + L_i, and denote this difference by Δ,

$$\Delta = \mathrm{EF}(T_i) - \mathrm{LF}(T_i) \qquad (5)$$

The method to translate the SGP$_{t\text{-}s}$ to the SGP$_{s\text{-}s}$ simply involves adding Δ to the ES time of T_t. An ES time analysis is then conducted again on the graph nodes contained in the paths dependent on T_t. After completing this ES time adjustment, an LF time analysis is required as before for all paths backward from the sinks. This process is repeated until no time conflicts are detected by equation (5); that is, $\Delta \leq 0$. The following algorithm determines both the LF times and the transient adjustments to the ES times and accounts for initial token transients as described above.

Given the linked-list graph representation shown in figure 3, a depth-first search algorithm that employs the same method used by the ES Algorithm (only in the reverse direction) will determine the latest finish times for all nodes (tasks). The algorithm calls the procedure SearchBkwd recursively for each input edge. As with the ES Algorithm, the recursive nature of this backward-search algorithm results in a depth-first search of a graph from sinks to sources while implicitly retaining the next edge (starting point for the next path to traverse when fanning out) and accumulated path latency on the memory stack. The arguments passed in to SearchBkwd are an address pointer (edge) to an edge object in figure 3 and a latency value (path_latency). This latency value is defined as the TBIO at the starting sink less the sum of node latencies along the current path from the sink up to an encountered node. As in the SearchFwd procedure, let node specify a pointer to a node structure of figure 3. An edge will point to a next_input if present, and will be null if no other input edges for the current node exist. The iterative nature of the LF Algorithm for the cases where initial tokens are present within the DFG requires the inclusion of a boolean condition. The boolean condition Done in the LF Algorithm indicates when the process of determining LF times for all nodes is complete. The LF Algorithm is stated as follows:

A. Initialize all LF times of tasks in $\mathcal{T}$ to maximum storage value and set Done = False.

B. While not Done Loop through to Step K.

C. Set Done to True and repeat Step D for every sink in the graph.

D. If the sink is not virtual, set LF equal to the earliest start of the sink (already established by the ES Algorithm) and skip to Step J; else determine the terminal node, T_t, of the edge with the initial token and set LF equal to ES(T_t) + d(TBO) where ES(T_t) is the earliest start of T_t, d is the number of initial tokens, and TBO is the iteration period.

E. Set Δ equal to earliest finish of T_i minus LF.

F. If Δ is less than or equal to zero go to Step J; else set Done to False.

G. Increase the earliest start of T_t by Δ.

H. Call the procedure SearchFwd (T_t.output, ES(T_t) + L_t) of the ES Algorithm in order to propagate the Δ time shift for all descendent nodes of T_t.

I. Increase LF by Δ.

J. Call the procedure SearchBkwd (sink. input, LF).

K. Loop until Done.

SearchBkwd (edge, path_latency)

1. If edge.next_input is not null, call SearchBkwd (edge.next_input, path_latency).
2. Get the node that uses this edge for output by setting node equal to edge.initial_node.
3. Determine the latest finish of node, LF (node), such that LF (node) = min [LF (node), path_latency].
4. Decrease path_latency by the node latency, L_{node}.
5. Set edge equal to the first input edge of node, edge = node.input.
6. If a source has been reached (edge = null), return from this procedure; else repeat Step 1.

Since the method just presented to translate the SGP$_{t\text{-}s}$ to the SGP$_{s\text{-}s}$ is recurrent, one may question if a solution exists for all cases. This is important since, if a solution does not exist, the method would hang in an infinite loop. The answer is yes, there is a solution. The proof lies in the fact that the only potential problem results when circuits with initial tokens are present in the dataflow graph. If adjustments were made to the ES times of the nodes dependent on the edge initialized with tokens that eventually led back to the original edge (due to a circuit) with a new EF time, the new EF time would again cause a conflict in equation (4), and the process would repeat indefinitely—a run-away condition. Such a condition implies that nodes firing on tokens propagating through such a circuit could not produce a token on the initialized edge in a timely manner. It has been shown that the minimum graph-theoretic iteration period, T_0, is given by the ratio of the ith circuit latency, C_i, to the number of tokens in the circuit, D_i for all circuits within the DFG (refs. 3, 9, 11, and 14):

$$T_o = \max\left(\frac{C_i}{D_i}\right) \qquad \text{(for all } i\text{th circuits)} \qquad (6)$$

Equation (6) determines the minimum time in which tokens can propagate through a circuit in one periodic cycle and thus establishes a lower bound on TBO. The only way this algorithm would fail to complete is if the TBO of equation (4) is less than its lower bound T_0 given by equation (6). Since TBO cannot be less than T_0, such a timing conflict cannot occur and thus the ES/LF algorithms previously presented will always have a solution.

As an alternative approach, the steady-state ES times could be determined during the forward search of the graph by applying equation (4) (solving for ES(T_t) with LF(T_i) set equal to the path latency) whenever encountering forward-path initial tokens. After determining all steady-state ES times, the LF times could then be calculated without requiring any further adjustments to the ES times, resulting in a one-time pass of the graph in the forward and backward direction. The algorithms are presented in the potentially recurrent form for the purpose of efficiently handling the frequent cases. That is, application of equation (4) (solved for ES(T_t)) would be required each time an edge with initial tokens was encountered by traversing multiple paths that may converge on the edge. Use of equation (4) once when beginning with a virtual sink would tend to minimize its use. Also, it is felt that the frequent cases involve uninitialized edges or initialization of recurrence loops (no forward-path tokens). Thus, this only requires the one-time use of equation (4) by the `LF Algorithm` for the purpose of calculating slack time within the recurrence loop. Like the `ES Algorithm`, the time complexity of the `LF Algorithm` is bounded by equation (1). Thus, the `LF Algorithm` can also be executed in polynomial time with a worst-case bound of $O(N^3)$.

Applying the `LF Algorithm` to the DFG of figure 1 for a TBO of 333 clock units is shown in figure 8. As expected, the slack time of task C extends all the way to the start time of task F. This would also be the case for task E if it were not for the initial token on the E $\prec$ D edge. Because of this token, the slack time of task E extends out only 33.3 clock units for the current iteration period of 333 clock units. The fact that this slack is associated with the next iteration of task D is apparent from the TGP diagram of figure 5 where the time between the completion of task E and the start of task D is equal to 33.3 clock units.

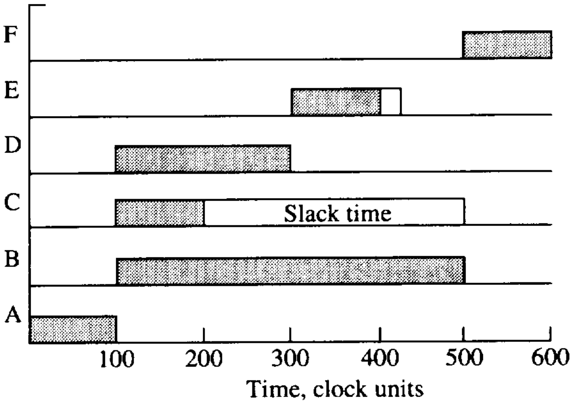


Figure 8. Single graph play diagram showing slack time. ω = 600 clock units.

4. Performance Metrics and Resource Requirements

The two types of concurrency that can be exploited in dataflow algorithms can be classified as parallel and pipeline. The TBO and TBIO performance metrics defined in the previous sections are important in evaluating the efficiency of the algorithm execution, that is, how well the inherent parallelism within the algorithm is being exploited. Therefore, it is important to determine the bounds on these metrics which define the optimum scheduling solution.

4.1. Critical Path Analysis

Parallel concurrency is associated with the execution of tasks that are independent (no precedence relationship imposed by $\prec$). The extent to which parallel concurrency can be exploited is dependent on the number of parallel paths within the DFG and the number of resources available to exploit the parallelism. The TBIO metric in relation to the time it would take to execute all tasks sequentially can be a good measure of the parallel concurrency inherent within a DFG. If there are no initial tokens present in the DFG, TBIO can be determined with the traditional critical path analysis, where TBIO is given as the sum of latencies in $\mathcal{L}$ along the critical path. When $\mathcal{M}_0$ defines initial tokens in the forward direction, the graph takes on a different behavior as represented by the new paths within the MDFG. Cases such as this include many signal processing and control algorithms where initial tokens are expected to provide previous state information (history) or to provide delays within the algorithm. For the example shown in figure 9, the task output $z(n)$ associated with the nth iteration is dependent

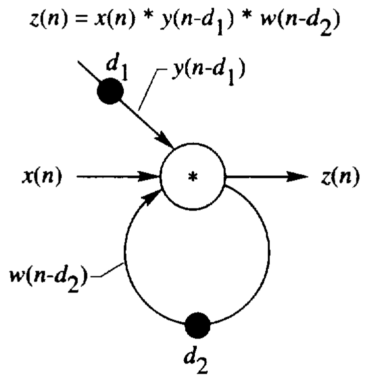


Figure 9. Example function implementation.

on the current input $x(n)$, input $y(n-d_1)$ provided by the $(n-d_1)$th iteration, and input $w(n-d_2)$ produced by the $(n-d_2)$th iteration.

Implementation of this function would require d_1 initial tokens on the $y(n-d_1)$ edge and d_2 initial tokens on the $w(n-d_2)$ edge in order to create the desired delays. In such cases, the critical path and thus TBIO are also dependent on the iteration period TBO. For example, given that a node fires when all input tokens are available, assuming sufficient resources, the earliest time at which the node shown in figure 9 could fire would be dependent on the longest path latency leading to either the $x(n)$ or $y(n-d_1)$ edge. Assuming that the d_1 and d_2 tokens are the only initial tokens within the graph, the time it would take a token associated with the *n*th iteration to reach the $x(n)$ edge would equal the path latency leading to the $x(n)$ edge. Likewise, the minimum time at which the "token" firing the *n*th iteration on the $y(n-d_1)$ edge could arrive from the source equals the path latency leading to the $y(n-d_1)$ edge. However, since this "token" is associated with the $(n-d_1)$th iteration (produced d_1 (TBO) intervals earlier), the actual path latency referenced to the same iteration is reduced by the product of d_1 and TBO. From this example, it is easy to infer that the actual path latency along any path with a collection of *d* initial tokens is equal to the summation of the associated node latencies less the product of *d* and TBO. Thus, the critical path (and TBIO) is a function of TBO and is given as the path from source to sink that maximizes the following equation for TBIO:

$$\text{TBIO} = \max\left[\left(\sum_{i \in \mathcal{L}} L_i\right) - d(\text{TBO})\right] \quad \text{(for all paths)} \tag{7}$$

where *d* is the total number of initial tokens along the path. It is easy to see that the critical path for the DFG in figure 1 is $A \prec B \prec F$, resulting in a TBIO of 600 clock units.

4.2. Calculated Speedup

Pipeline concurrency is associated with the repetitive execution of the algorithm for successive iterations without waiting for earlier iterations to complete. Equation (6) defines the lower bound iteration period T_o due to the characteristics of the graph alone. That is, if circuits are present in the DFG, T_o is given by equation (6), otherwise T_o is zero. Given a finite number of processors, however, the actual lower bound on iteration period (or TBO_{lb}) is given by

$$\text{TBO}_{lb} = \max\left(T_o, \left\lceil \frac{\text{TCE}}{R} \right\rceil\right) \tag{8}$$

where TCE (total computing effort) is the sum of latencies in $\mathcal{L}$,

$$\text{TCE} = \sum_{i \in \mathcal{L}} L_i \tag{9}$$

and *R* is the number of available processors. The theoretically optimum value of *R* for a given TBO period, referred to as the calculated *R*, is given as

$$R = \left\lceil \frac{\text{TCE}}{\text{TBO}} \right\rceil \tag{10}$$

Since every task executes once within an iteration period of TBO with *R* processors and takes TCE amount of time with one processor, speedup *S* using Amdahl's Law can be defined as

$$S = \frac{\text{TCE}}{\text{TBO}} \tag{11}$$

and processor utilization *U* ranging from 0 to 1 can be defined as

$$U = \frac{S}{R} \tag{12}$$

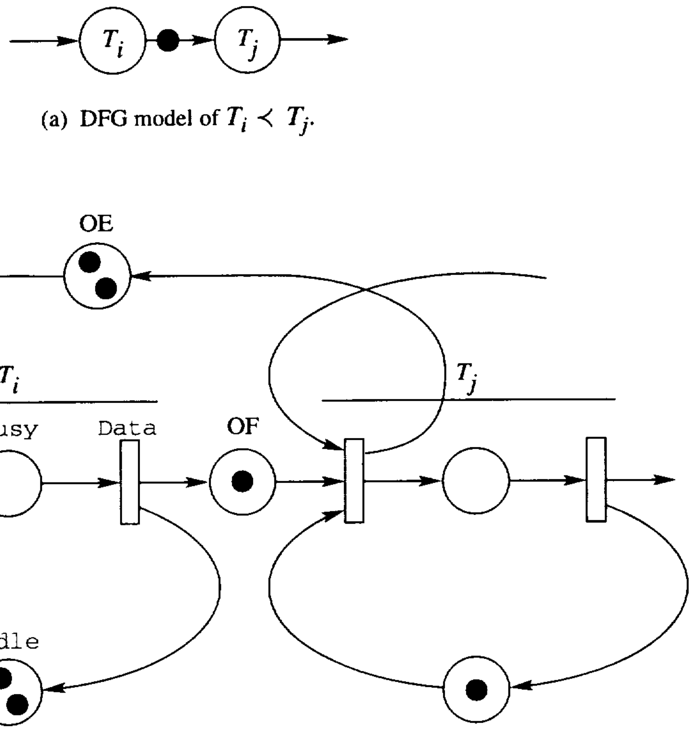
4.3. Run-Time Memory Requirements

The scheduling techniques offered by this paper are intended to apply to the periodic execution of algorithms. In many instances, the algorithms may execute indefinitely on an unlimited stream of input data, for example, digital signal processing algorithms. Even though the multiprocessor schedules determined by the `ES Algorithm` and `LF Algorithm` are periodic, it is important to determine if the memory requirements for the data are bounded. Just knowing that the memory requirements are bounded may not be enough. One may also wish to calculate the maximum memory requirements a priori. By knowing the upper bound on memory, the memory can be allocated statically at compile time to avoid the run-time overhead of dynamic memory management.

Since the dataflow graph edges imply physical storage of the data shared among tasks, graph-theoretic rules are defined in this section capable of determining the bound on memory required for the shared data.

To present a slightly more detailed model of parallel computation of tasks represented by a DFG is helpful for the following discussion. The Petri net model shown in figure 10 describes the activities associated with the execution of ordered dataflow tasks, $T_i \prec T_j$. A Petri net such as the one shown in figure 10 is a special class of Petri nets called a marked graph (ref. 15). This model is equivalent to the ATAMM computational marked graph (CMG) shown in references 13, 14, and 16. As shown in figure 10, the edges directed from left to right represent dataflow while the edges from right to left represent control flow. Of particular interest, the edges associated with the output empty (OE) place can be regarded as an "acknowledgment edge." That is, given the data dependency $T_i \prec T_j$, the acknowledgment edge provides a signal to node T_i indicating that node T_j has consumed a token from the output full (OF) place. The number of tokens present at any one time in the OE place represents the total number of empty data buffers available for output data tokens. The number of buffers currently occupied with data tokens is represented by the number of tokens in the OF place. Pairing every data edge with an acknowledgment edge assures that a buffer will be available for the output data before a task begins execution. A modeled task is enabled for execution when all necessary input tokens to the `Fire` transition are available. After firing, the node will produce a token in the `busy` place, enabling the `Data` transition. The `Data` transition for node T_i of $\mathcal{T}$ will generate a token at the output places after delaying an amount of time equal to L_i of $\mathcal{L}$. The `idle` place between the `Data` and `Fire` transitions is included to convey information about task instantiations at run time. The graph shown in figure 10(b) has been shown to be consistent (refs. 11 and 15). This implies that given an initial marking, the total number of tokens within a circuit remains unchanged for all valid markings reached by firing transitions. Therefore, the initial number of tokens located in the `idle` place will ultimately migrate to the `busy` place; this indicates the number of task instantiations at run time. Based on equation (6), the number of tokens that must be present in a circuit for a given iteration period, TBO, is given by the following equation

$$D_i = \left\lceil \frac{C_i}{\text{TBO}} \right\rceil \qquad \text{(for all } i \text{ circuits)} \qquad (13)$$



(a) DFG model of $T_i \prec T_j$.

(b) Petri net model.

Figure 10. Petri net representation of dataflow graph.

and thus the circuit formed by the `idle` place between the `Data` and `Fire` transition implies that the required number of instantiations of task T_i that was derived from the TGP diagram is determined by the following equation:

$$\text{Instantiations of } T_i = \left\lceil \frac{L_i}{\text{TBO}} \right\rceil \tag{14}$$

Because DFG tokens carry data values (or pointers to where the data are located when the tokens become heavy), the DFG edges which transport tokens from one node to the next, imply physical memory space. Again relying on the token conservation property, the summation of the initial OF tokens due to initial data and the initial number of OE tokens needed to satisfy equation (13) determines the maximum buffer space required for the data associated with the DFG edge at run time—ideally, ignoring fault tolerant issues. The initial tokens required in the OE and OF places can also be determined from the TGP diagram, but in a less obvious way.

Initial OE tokens can be determined by examining the relative firing times of the predecessor and successor tasks along with the corresponding data set displacements. The OE Rule can be used to determine the initial number of OE tokens indicating the data buffers that are initially empty and is as follows:

> Let $S(T_i)$ represent the start time of task T_i relative to a TBO interval as portrayed in the TGP diagram, and let $D_s(T_i)$ represent the relative data set number associated with the start time of task T_i. The start time $S(T_i)$ can be calculated directly from the ES of T_i with the equation
>
> $$S(T_i) = \text{ES}(T_i) \text{ modulo TBO} \tag{15}$$
>
> The relative data set number can also be determined from the TGP diagram or calculated directly by the equation
>
> $$D_s(T_i) = \mathcal{P} - \left\lfloor \frac{\text{ES}(T_i)}{\text{TBO}} \right\rfloor \tag{16}$$
>
> where the floor function is applied to the ratio of ES(T_i) and TBO, and $\mathcal{P}$ is given by equation (2). Then, given a task T_p, let T_s represent the successor task which uses the output data of T_p as input and OE$_{\text{ps}}$ be the initial OE tokens required for the precedence relation $T_p \prec T_s$.

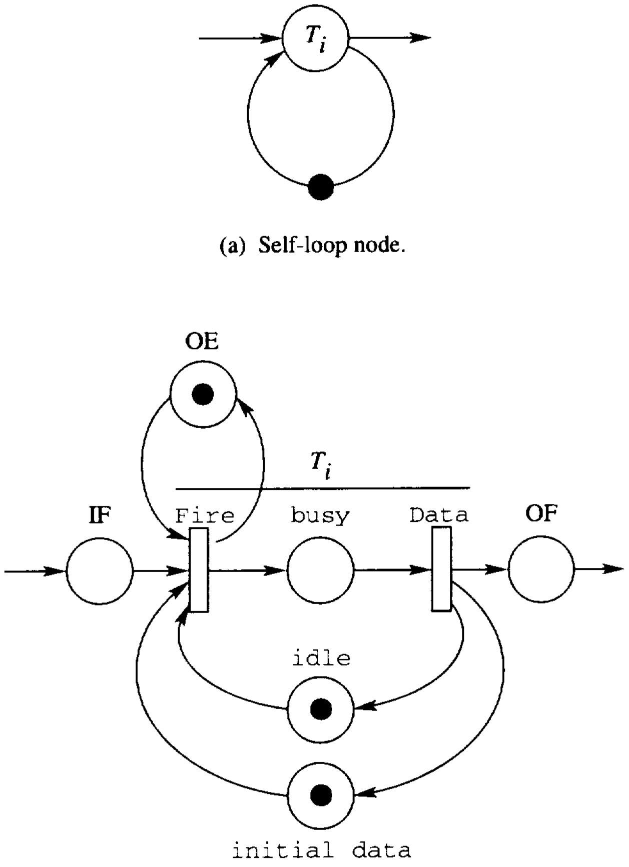
```
If $D_s(T_p) - D_s(T_s) \geq 0$
Then If $S(T_p) \leq S(T_s)$
    Then $\text{OE}_{\text{ps}} = D_s(T_p) - D_s(T_s) + 1$
    Else $\text{OE}_{\text{ps}} = D_s(T_p) - D_s(T_s)$
Else $\text{OE}_{\text{ps}} = 0$
```

In terms of the graph nodes, a negative $D_s(T_p) - D_s(T_s)$ indicates that the successor node has fired more often than the predecessor node it is dependent on. The only way this could be possible is if there were initial tokens present in the OF place. A positive difference $D_s(T_p) - D_s(T_s)$ represents the number of times the predecessor node fires before the successor node fires once. This difference would therefore be the initial tokens required in the OE place. If $S(T_p) > S(T_s)$ then the successor node would have returned the one token required in the OE place for the predecessor to fire again, and thus no additional tokens are needed. However, the condition $S(T_p) \leq S(T_s)$ indicates that the predecessor node must fire before or at the same time the successor node fires and returns the OE token. Therefore, the $S(T_p) \leq S(T_s)$ condition requires that one extra token be included initially in the OE place.

For example, the `OE Rule` utilizing the TGP of figure 5 for the C $\prec$ F specifies that OE$_{\text{CF}} = 2$ or in other words, two empty data buffers are initially required. Since the data edge did not have any initial tokens (no initially full buffers), two buffer spaces would be required at run time.

There is one item that must be mentioned concerning the `OE Rule`. For all practical purposes the $\leq$ in the $S(T_p) \leq S(T_s)$ expression can be replaced with a $<$. This change has the effect of delaying the firing of the predecessor node by one `Fire` transition time when T_p and T_s would otherwise start simultaneously. If the `Fire` transition time which may represent the reading of input data is considered negligible in the case of large-grained algorithms, being conservative with tokens (and thus buffer space) is easily tolerated. The rule represents the more conservative case in order to satisfy the general problem. One special case is shown in figure 11 as a node with a self-recurrence circuit (representing the fact that the task represented by the node has history). The `OE Rule` would indicate that one initially empty buffer is needed in addition to the initial data occupying a second buffer. Use of the conservative token approach would not make sense in this case because a node that is self-dependent cannot wait on itself to fire.

The `OE Rule` determines the number of data buffers needed *in addition* to the buffers required for initial data for all edges within the DFG. Therefore, the resource requirements in terms of total buffer space for a given data edge is equal to the OE tokens given by the `OE Rule` plus the number of initial tokens present on the edge. Calculating resource requirements in terms of processors is more straightforward. The minimum processor requirement R for a given TBO at steady-state can be derived simply by counting the maximum overlap of bars within the corresponding TGP. However, the R

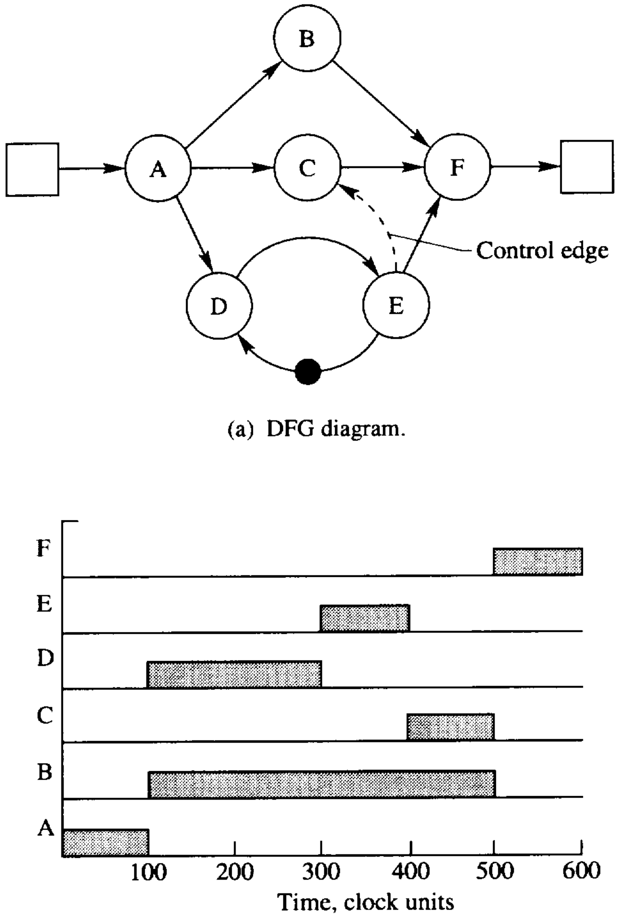


(a) Self-loop node.

(b) Petri net model.

Figure 11. Petri model of self-loop circuit.

determined may not be optimum for a given $\prec$. For example, given only three processors, TBO$_{lb}$ for the DFG of figure 1 by equation (8) is equal to 333, which by equations (11) and (12) would indicate that three processors would provide maximum linear speedup with 100 percent processor utilization. Even though the processor requirements for a single graph iteration is three (determined by counting the maximum overlap of bars in fig. 8), the processor requirements for repetitive execution with a period of 333 requires four processors as can be derived from figure 5. This is because of the fact that the precedence constraints imposed by $\prec$ makes finding this optimal solution NP-complete and the design process presented in this paper only provides the determination of a *sufficient* number of processors in order to guarantee a schedule meeting TBO and TBIO requirements (refs. 9 and 10). In fact, one cannot guarantee that a multiprocessor-scheduling solution even exists when all three parameters (TBO, TBIO, and R) are fixed (ref. 9). Accordingly, it is necessary to find another schedule, if one exists, that would provide the desired computational speedup performance; a method for doing so is discussed in the next section.



(a) DFG diagram.

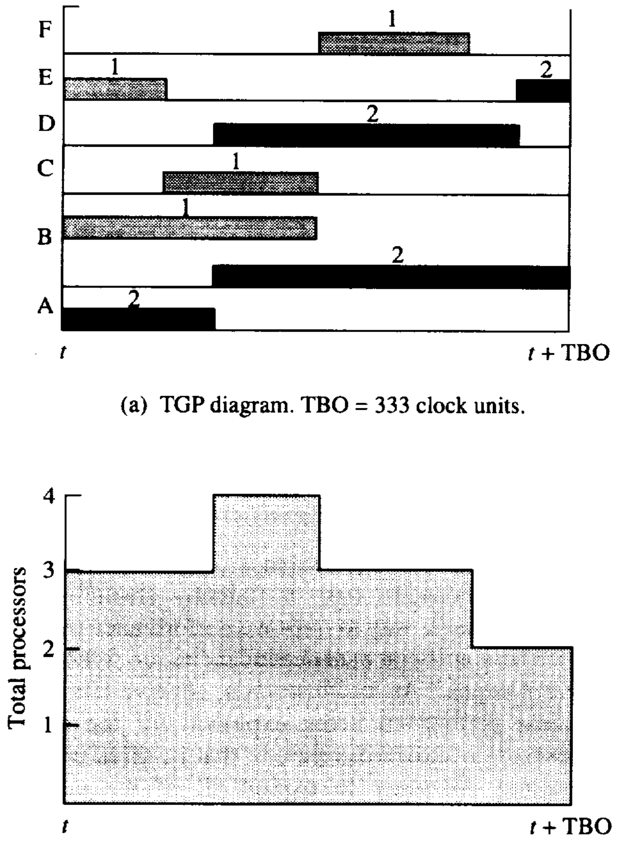
(b) SGP diagram. ω = 600 clock units.

Figure 12. Diagrams with E $\prec$ C control edge.

4.4. Control Edges

Imposing additional precedence constraints or artificial data dependencies onto $\mathcal{T}$ (thereby changing the schedule) is a viable way to improve performance (refs. 5 and 17). These artificial data dependencies are referred to as "control edges." As an illustration, observe that there is needless parallelism being exploited for the single graph execution shown in figure 8; that is, three processors are not necessary to exploit all of the parallel concurrency—two would suffice. This presents an opportunity to take advantage of the slack time present in the graph to reduce the processor requirement without affecting the critical path.

Since task C does not need to complete execution until 500 clock units as shown in figure 8, a control edge can be included in order to create the precedence relationship E $\prec$ C effectively delaying task C until the completion of task E as shown in figure 12. The subsequent TGP with the added control edge is shown in figure 13 with the resulting resource envelope showing the processor utilization over the given TBO period. As can be seen from figure 13, it is only necessary to effectively move the amount of effort requiring four processors in such a way as to fill the idle time shown in the resource envelope. It turns out in this example that this can be done by delaying task D behind task B (a delay of 67 clock units) in relation to the TGP description of steady-state behavior. The new TGP diagram can be derived from the original by shifting all successor tasks of task D accordingly. The TGP diagram with the added B $\prec$ D precedence relationship shown in figures 14(a) and (b) results in 100 percent processor utilization. The new steady-state SGP shown in figure 14(c) can be constructed by shifting tasks D, E, C, and F to the right by 67 clock units, as was done to obtain the new TGP diagram.



(a) TGP diagram. TBO = 333 clock units.

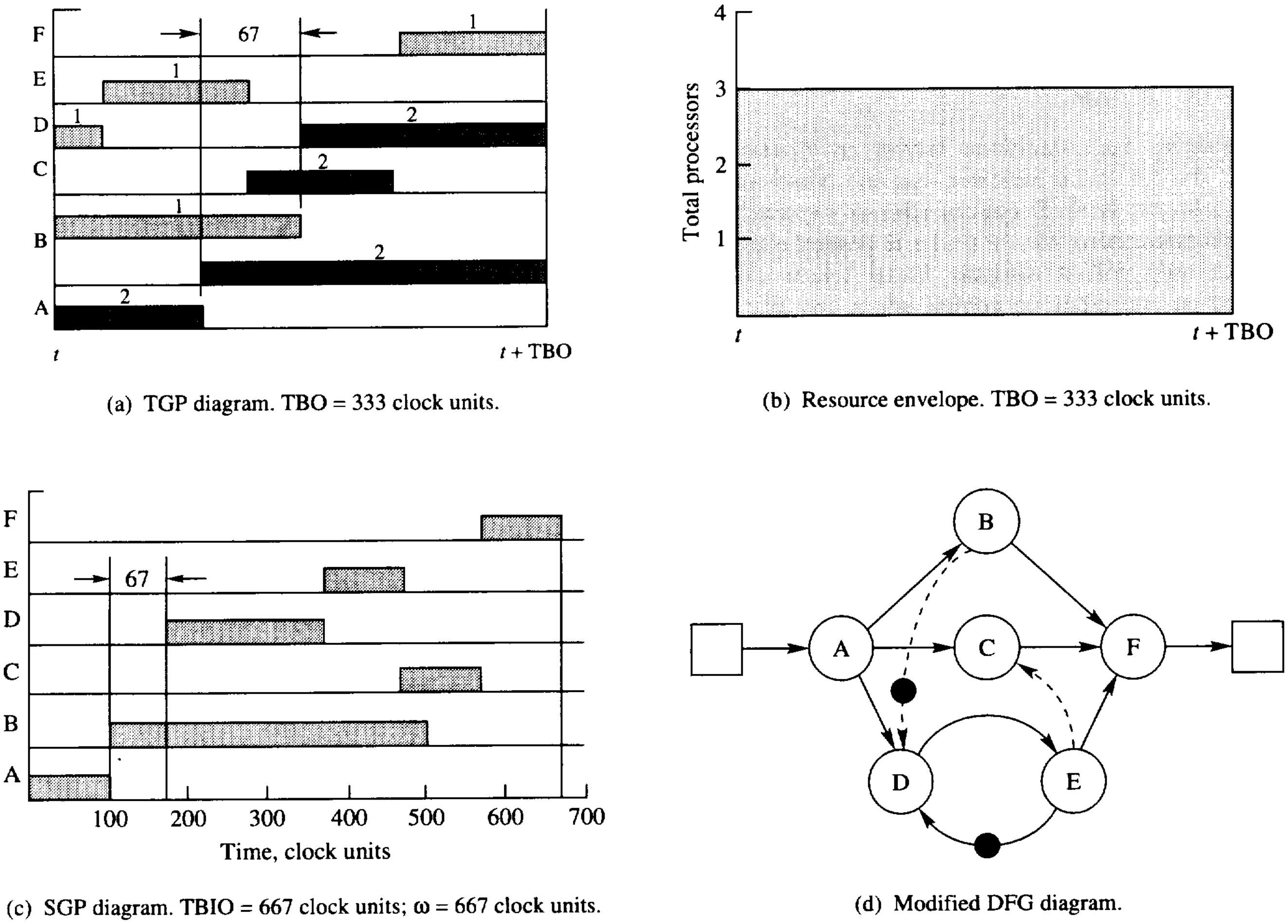
(b) Resource envelope.

Figure 13. Periodic behavior with E $\prec$ C control edge.

Referring to the new SGP diagram in figure 14(c), it is apparent that this scheduling solution for optimum throughput and processor utilization has been achieved at the cost of increasing TBIO. Inserting the B $\prec$ D precedence relationship to delay the start of task D behind the start of task B by 67 clock units, resulting in a TBIO of 667 clock units, is an interesting concept. Since we know that three processors are sufficient for tasks B and D to start at the same time for the first iteration, the B $\prec$ D precedence relationship has caused a transient condition. The reason for this transient becomes apparent by examining the TGP schedule of figures 14(a) and (b). The TGP schedule indicates that the *n*th token (relative data set number 2) consumed by node D is the $(n-1)$th token (relative data set number 1) produced by the predecessor node B; this implies that one initial token is required on the B $\prec$ D control edge, as shown in figure 14(d), to create the single-TBO delay required to achieve the steady-state schedule shown in figures 14(a) and (b). Without the single-TBO synchronization delay due to the initial token, the path A $\prec$ B $\prec$ C $\prec$ D $\prec$ E $\prec$ F would result in a TBIO equal to the graph TCE of 1000 rather than 667 clock units (eq. (7)). This is interesting in that the transients caused by initial data token delays that tend to complicate the analysis become a useful trait for control edges. Without initial tokens, control edges have only *intra-iteration* precedence relationships between two tasks and consequently provide only limited rescheduling options. The rescheduling options are those shown by the SGP diagram between independent tasks. Control edges properly initialized with tokens result in *inter-iteration* relationships between tasks that provide additional rescheduling options. Such control edges allow one to choose rescheduling options from the TGP diagram which can provide more opportunities to find tasks to delay behind other tasks.

Up to now, a general rule for calculating OF tokens was not needed because the initial data tokens are given by the algorithm description as portrayed in figure 9. However, with the use of control edges it is necessary to calculate the required number of OF tokens. The question that may have been raised about the OE Rule is what if $D_s(T_p) - D_s(T_s)$ is a negative number; this would mean that the tokens bounded to this edge circuit are initially located in the OF place. Just like any linear algebra problem with two unknowns, two rules (equations) are required in order to solve for the total number of tokens (OE and OF) needed within a given edge circuit. This second rule is referred to as the "OF Rule" and determines the number of tokens, if any, initially required on the forward (OF) edge. The OF Rule is stated as follows:

> Let $S(T_i)$ and $F(T_i)$ represent the start time and finish time of the tasks T_i, respectively, and let $D_s(T_i)$ represent the relative data set number associated with the start of task T_i; $S(T_i)$,



(a) TGP diagram. TBO = 333 clock units.

(b) Resource envelope. TBO = 333 clock units.

(c) SGP diagram. TBIO = 667 clock units; ω = 667 clock units.

(d) Modified DFG diagram.

Figure 14. Periodic behavior with E $\prec$ C and B $\prec$ D control edges.

$F(T_i)$, and $D_s(T_i)$ are relative to a TBO interval as portrayed in the TGP diagram. As for the OE Rule, these values can be obtained from the TGP diagram or from equations (15) and (16) with the addition of

$$F(T_i) = (\mathrm{ES}(T_i) + L_i) \text{ modulo TBO} \quad (17)$$

Because the data set number associated with the start of execution will be greater than the data set number associated with the completion of a multiply-instantiated task, let $D_f(T_i)$ represent the relative data set number associated with the finish time of task T_i, which can be calculated with

$$D_f(T_i) = \mathcal{P} - \left\lfloor \frac{\mathrm{ES}(T_i) + L_i}{\mathrm{TBO}} \right\rfloor \quad (18)$$

Then, given a task T_p, let T_s represent the successor task which uses the output data of T_p as input and $\mathrm{OF}_{\mathrm{ps}}$ be the initial OF tokens required for the precedence relation $T_p \prec T_s$.

```
If $D_s(T_s) - D_f(T_p) \geq 0$
Then If $S(T_s) < F(T_p)$
    Then $\mathrm{OF}_{\mathrm{ps}} = D_s(T_s) - D_f(T_p) + 1$
    Else $\mathrm{OF}_{\mathrm{ps}} = D_s(T_s) - D_f(T_p)$
Else $\mathrm{OF}_{\mathrm{ps}} = 0$
```

In terms of the graph nodes, a negative $D_s(T_s) - D_f(T_p)$ indicates that the predecessor node has fired more often than its successor node which is the frequent case. These tokens are accounted for in the OE Rule. A difference $D_s(T_s) - D_f(T_p) \geq 0$ represents the number of times the successor node fires before the predecessor node completes just once. The only way this could occur is if there were initial tokens in the OF place. This difference would therefore be the number of initial tokens required in the OF place. If $S(T_s) \geq F(T_p)$, then the predecessor node would have deposited the one token required in the OF place for the successor node to fire again, and thus no additional tokens are needed. However, the condition $S(T_s) < F(T_p)$ indicates that the successor node must fire before the predecessor node deposits an OF token. Therefore, the $S(T_s) < F(T_p)$ condition requires that one extra token be included initially in the OF place.

Applying the conditions shown in figures 14(a) and (b), the `OF Rule` indicates that one initial token is required on the B $\prec$ D control edge as expected from this discussion. Also, the `OF Rule` is general enough so that not only will it compute initial tokens (if any) required on inter-iteration control edges, but also agree with initial token conditions on data edges in most cases. In some cases, initial data tokens may only serve the purpose for which they were intended, that is, to create delay conditions for computations as portrayed in figure 9. When initial data tokens also affect the steady-state schedule, the `OF Rule` applied to such data edges would agree with the initial conditions. Just such a case involves the E $\prec$ D in the example graph. As one would expect, the `OF Rule` utilizing the TGP of figure 5 for the E $\prec$ D edge results in $\mathrm{OF}_{\mathrm{ED}} = 1$, which indicates that one initial token is present. Likewise, the `OE Rule` specifies that $\mathrm{OE}_{\mathrm{ED}} = 0$ indicates that an initially empty buffer is not necessary at run time, thereby the total buffer space for edge E $\prec$ D is defined as 1. However, just as the primary purpose of the `OE Rule` is to compute the number of data buffers required in addition to the initial data buffers, the primary purpose of the `OF Rule` is to compute initial tokens for inter-iteration control edges. The `OF Rule` applied to data edges will only convey information that the user already knows. Likewise, since by definition, control edges do not require data buffers, the `OE Rule` does not serve a purpose for control edges unless for some reason the user wanted to implement a graph management operating system that treated data edges and control edges the same, except for the attachment of physical buffers.

One last example would be appropriate before presenting the Design Tool which implements the algorithms and rules discussed in this and previous sections. It has been shown that the addition of the E $\prec$ C and B $\prec$ D control edges for a TBO of 333 clock units results in linear speedup with three processors and a TBIO equal to 667 clock units. Since this particular solution includes an initial token in the forward direction of the B $\prec$ D edge, analyzing this graph with the `ES and LF Algorithms` should confirm the correctness of the solution. The modified dataflow graph of figure 14(d) with the additional control edges is shown in figure 15. Utilization of the `ES Algorithm` results in earliest start times of ES(A) = 0, ES(B) = ES(A) + $\mathcal{L}$(A) = 100, ES(D) = ES(A) + $\mathcal{L}$(A) = 100, ES(E) = ES(D) + $\mathcal{L}$(D) = 300, ES(C) = ES(E) + $\mathcal{L}$(E) = 400, and ES(F) = ES(C) + $\mathcal{L}$(C) = ES(B) + $\mathcal{L}$(B) = 500.

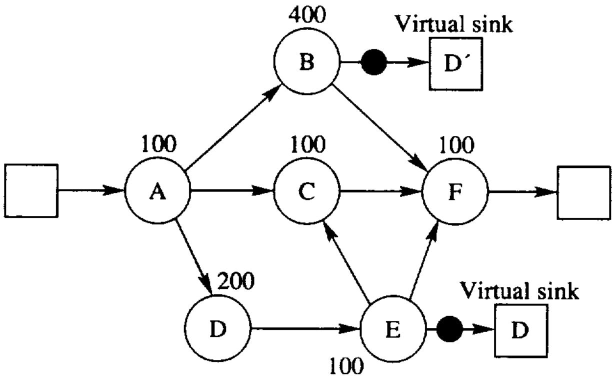


Figure 15. Equivalent MDFG model of figure 14(d).

The first application of the backward search by the `LF Algorithm` beginning at the real sink results in latest finish times of LF(F) = ES(sink) = EF(F) = 600, LF(B) = LF(F) − $\mathcal{L}$(F) = 500, LF(A) = LF(B) − $\mathcal{L}$(B) = 100, LF(C) = LF(B) = 500, LF(E) = min[LF(C) − $\mathcal{L}$(C), LF(F) − $\mathcal{L}$(F)] = 400, LF(D) = LF(E) − $\mathcal{L}$(E) = 300, and LF(A) = min[LF(B) − $\mathcal{L}$(B), LF(C) − $\mathcal{L}$(C), LF(D) − L(D)] = 100.

Next, applying the `LF Algorithm` beginning at the virtual sink (D') corresponding to the E $\prec$ D data edge gives an LF time for node E of ES(D) + (1)(TBO) = 100 + (1)(333) = 433 clock units which is greater than its earliest finish of 400 clock units. Progressing backwards does not change the latest finish times of nodes A and D. Finally, applying the `LF Algorithm` beginning at the virtual sink (D') corresponding to the B $\prec$ D control edge gives an LF time for node B of ES(D) + (1)(TBO) = 100 + (1)(333) = 433 clock units. However, since the previous ES analysis indicates that node B cannot complete until 500 clock units, a transient condition has been found with a Δ (eq. (5)) equal to 67 clock units. Therefore, node D initially starts execution as soon as node A completes during the transient state but at steady state, node D will be delayed after the completion of node A by 67 clock units. Adding Δ = 67 clock units to the ES time along the path D $\prec$ E $\prec$ C $\prec$ F results in adjusted earliest start times of ES(D)′ = 167, ES(E)′ = 367, ES(C)′ = 467, and ES(F)′ = 567.

Applying the `LF Algorithm` again at the virtual sink D' gives an LF(B) = ES(D)′ + (1)(333) = 500 equal to the earliest finish time of node B, as expected. After calculating the latest finish times once more, the steady-state scheduling contraints in terms of earliest start and latest finish times are defined. The TBIO is the earliest start of the sink and is determined to be EF(F) = ES(F)′ + $\mathcal{L}$(F) = 667 clock units. As a final check, the TBIO of 667 clock units should agree with

Table I. Summary of DFG Attributes for TBO = 333 clock units, TBIO = 667 clock units, and R = 3

Task	Latency	ES	LF	Instantiations	Output task	OE	OF	Total buffers
A	100	0	100	1	D	1	0	1
					C	2	0	2
					B	1	0	1
B	400	100	500	2	D	1	1	2
					F	2	0	2
C	100	467	567	1	F	1	0	1
D	200	167	367	1	E	1	0	1
E	100	367	467	1	C	1	0	1
					F	1	0	1
					D	0	1	1
F	100	567	667	1	Sink	1	0	1

equation (7) which finds the critical path. By equation (7), the path A $\prec$ B $\prec$ D $\prec$ E $\prec$ C $\prec$ F (containing all nodes and 1 initial token) has a total latency of (TCE – (1)(333)) = 667 clock units and is largest over all paths. Thus, the path A $\prec$ B $\prec$ D $\prec$ E $\prec$ C $\prec$ F is critical. Table I lists the steady-state earliest start and latest finish times obtained by applying the `ES` and `LF Algorithms` to the DFG of figure 15. The reader is invited to construct a single graph play diagram using the ES times in table I. Likewise, a total graph play diagram can be constructed by using start times equal to ES modulo TBO. The SGP and TGP should agree with figures 14(c) and (a), respectively.

A summary of other DFG attributes for the scheduling solution presented above is also provided in table I. The attributes listed include task instantiations, data memory requirements (buffers), and control edges for a TBO of 333 clock units, while utilizing three processors 100 percent of the time. As noted, this solution is optimum in terms of TBO and processor utilization but is not optimum in terms of TBIO. Note also that even though an optimum solution does not exist for this example where TBO, TBIO, and R are fixed to optimum values, depending on the real-time constraints of the application, one could have designed a solution which made other trade-offs in performance. For example, another solution might maintain a minimum TBIO of 600 clock units while letting TBO increase above the lower bound of 333 clock units. In general, depending on the availability of processors, the user has a two-dimensional region (TBO by TBIO) in which to make trade-offs. This region is referred to as an operating point plane in references 5 and 17; TBO_{lb} and TBIO_{lb} define the minimum values for the two dimensions, respectively.

5. Design Tool

The algorithms and rules presented in the previous sections have been shown to be applicable to the analysis of the class of dataflow graphs described in section 1. A software tool is presented in this section which analyzes dataflow graphs and implements these design principles to aid the user in the implementation of a multiprocessing application. The software, referred to as the "Dataflow Design Tool," or "Design Tool" for brevity, was written in Borland C++[2] for Microsoft Windows.[3] The software can be hosted on an i386/486 personal computer or compatible. The Design Tool takes input from a text file which specifies the topology and attributes of the DFG. A graph-entry tool has been developed to create the DFG text file. The various displays and features are shown to provide an automated and interactive design process which facilitates the selection of a multiprocessor dataflow solution.

The process flow of the Design Tool, upon loading a DFG or making modifications to the number of processors (R), iteration period (TBO), or adding control edges (new $\prec$), is shown in figure 16. After loading a DFG, the Design Tool will search the DFG for circuits in order to determine the minimum iteration period (T_o) using equation (6). The TBO will initially be set to the lower bound given in equation (8) where T_o is zero if no circuits are present. The calculated R will initially be given by equation (10). Next, the MDFG is automatically constructed due to initial tokens, if present, defined by the algorithm. All further analysis is based on the MDFG using the `ES Algorithm` and `LF Algorithm` in order to determine the TBIO, steady-state schedule ω and buffer requirements (using the `OE/OF Rules`). Any changes to TBO, R, or $\prec$ results in a reapplication of the analysis algorithms and rules.

The same dataflow graph example shown in figure 1 is used for demonstration purposes. In this way, the tool can be presented while verifying the theoretical results

[2] Version 3.1 by Borland International, Inc.

[3] Version 3.1 by Microsoft Corporation.

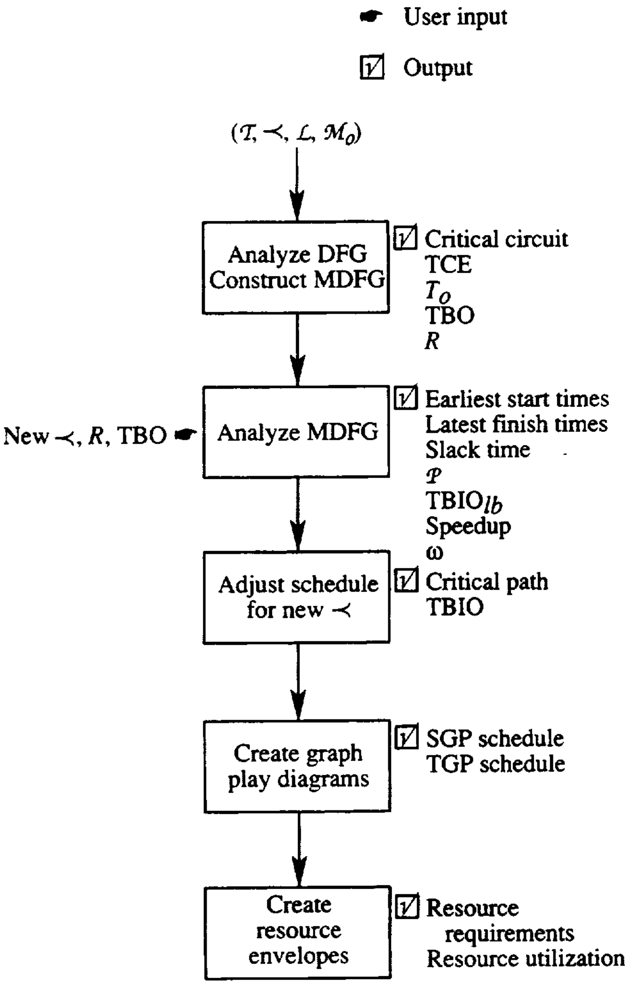


Figure 16. The design process.

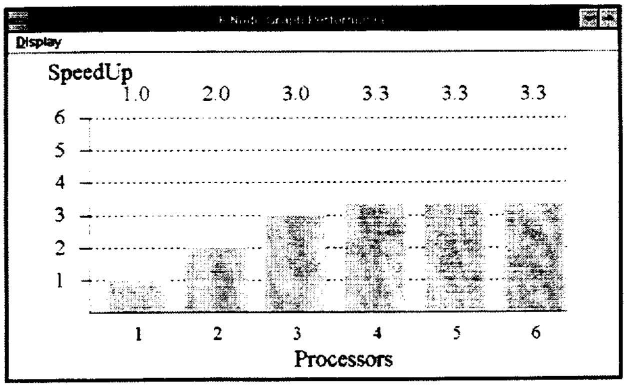
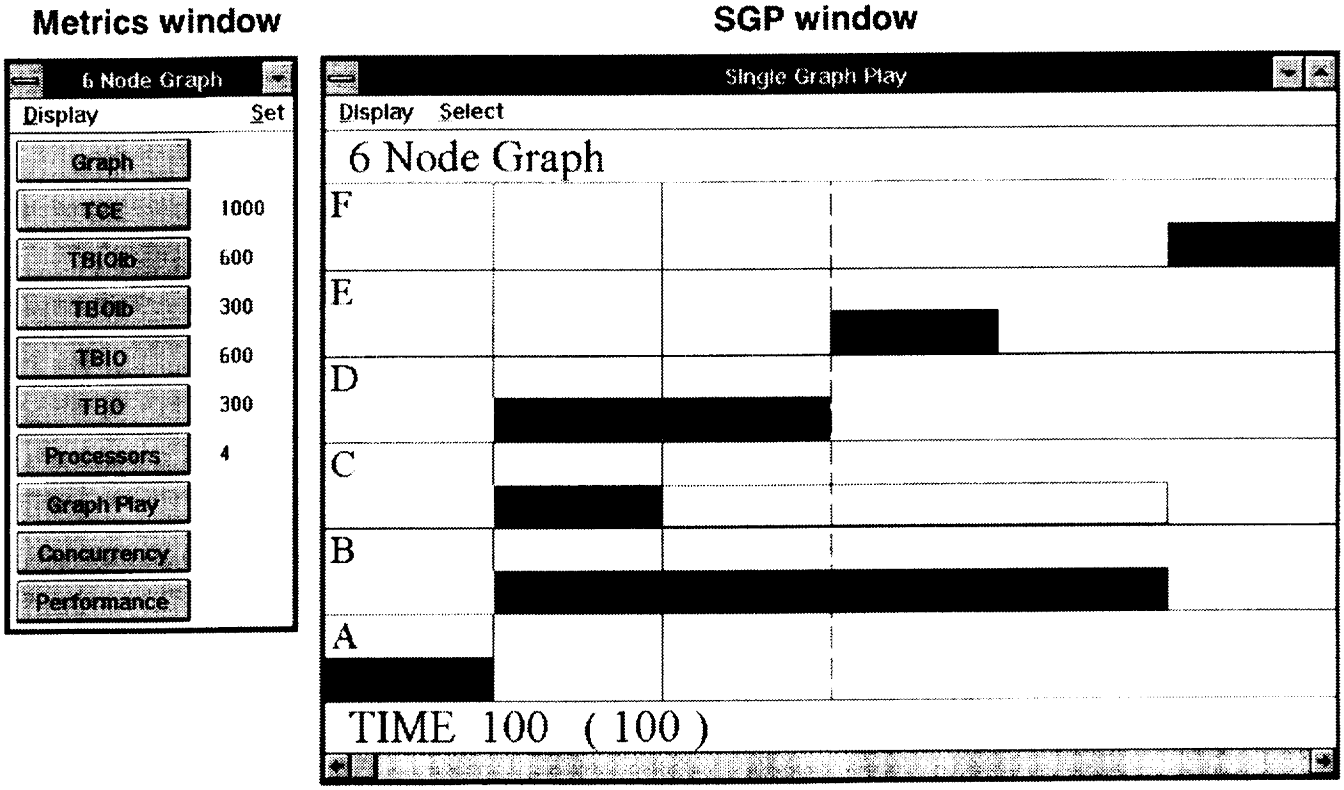


Figure 17. Speedup display.

obtained in the previous sections. The initial performance analysis, without any graph modifications, in terms of potential speedup is shown in figure 17 for up to six processors. The performance display shows speedup verses the number of processors. The display automatically increases or decreases the abscissa each time the number of processors R is changed. Figure 17 indicates that maximum speedup performance is attainable with four processors; additional processors will not result in any further speedup. This leveling-off of performance is attributable to the recurrence loop (circuit) within the DFG. Without this circuit, the graph-theoretic speedup would continue to increase linearly with the addition of processors. Physically speaking, however, this linear increase in speedup would ultimately break off due to operating-system overhead, such as synchronization costs and interprocessor communication.

The Design Tool has a user-interface panel, referred to as the "Metrics window" as shown in figure 18, containing buttons and menus for displaying performance bounds, setting TBO and R, or invoking the various graphic displays. For example, the display shown in figure 17 can be invoked by pressing the `Performance` button. The time measurements shown in the Design Tool windows are given in clock units so that the resolution of the measurement can be user interpreted. Upon analyzing the DFG, the Design Tool has determined that TCE is 1000 clock units. The TBIO_{lb} is defined by equation (7) based on the graph precedence relations $\prec$ due only to the data dependencies (dataflow). Due to the critical path A $\prec$ B $\prec$ F, TBIO_{lb} has been determined to be 600 clock units. The TBIO will be equal to TBIO_{lb} until additional control edges are added with the tool, which may change the critical path. The TBO_{lb} has been calculated to be 300 clock units based on the critical circuit D $\prec$ E, and consequently, TBO is set equal to this lower bound. The calculated R is determined to be 4, which is the optimum number of processors for repetitive, steady-state execution at the given TBO and TBIO.

The SGP window shown in figure 18, created by the Design Tool, shows the steady-state execution for a single iteration. The SGP window can be compared with that of figure 2. Slack time for task C is shown as an unshaded bar. Although there is slack between the completion of task E and the start of task F, the recurrence relation E $\prec$ D at a TBO of 300 clock units as determined by equation (4) has reduced the slack of task E to zero. The window also displays the two TBO-width segments with a vertical dashed line. Individually controlled left and right cursors (solid vertical lines) are provided for taking time measurements. Figure 18 shows the cursors measuring the start and duration time of task C to be 100 clock units each (the "100" next to time at the bottom of the display indicates the left-cursor time, whereas the "100" in parentheses indicates the time between the left and right cursors).



Metrics window

SGP window

6 Node Graph

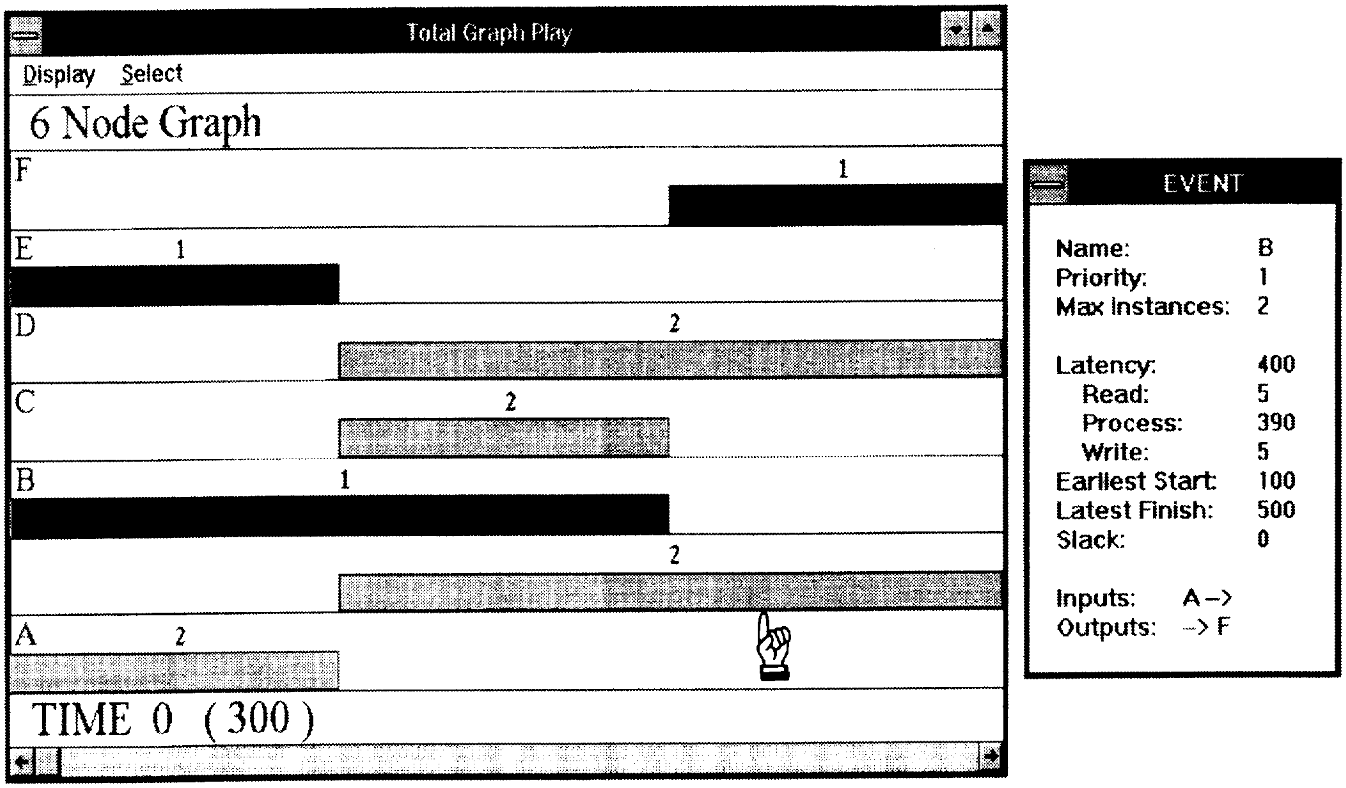
TCE	1000
TBIOlb	600
TBOlb	300
TBIO	600
TBO	300
Processors	4

Single Graph Play

6 Node Graph

TIME 100 (100)

Figure 18. Metrics and SGP window displays.



Total Graph Play

6 Node Graph

TIME 0 (300)

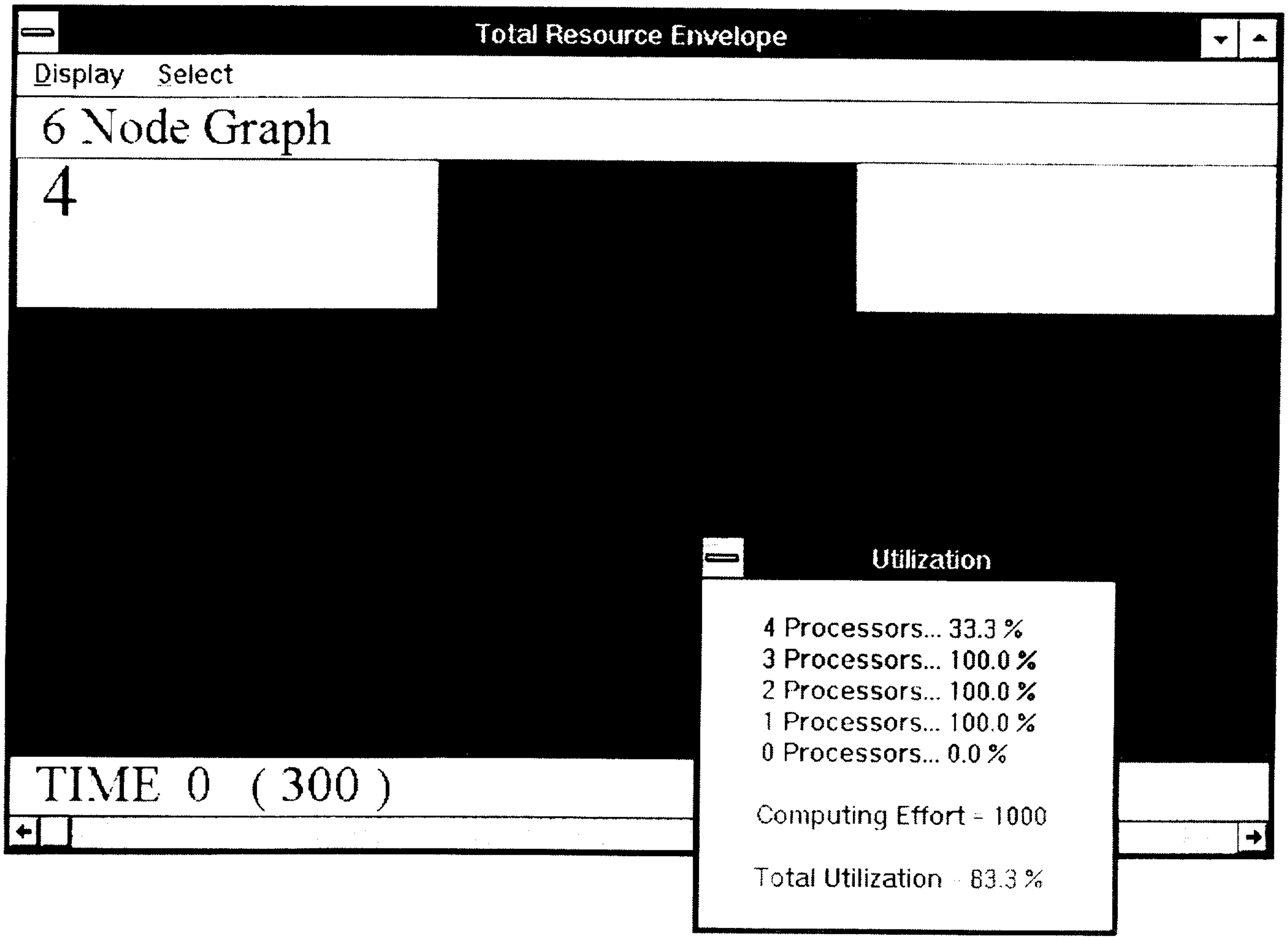
EVENT

Name:	B
Priority:	1
Max Instances:	2
Latency:	400
Read:	5
Process:	390
Write:	5
Earliest Start:	100
Latest Finish:	500
Slack:	0
Inputs:	A–>
Outputs:	–> F

Figure 19. TGP window.

The TGP window shown in figure 19 displays the steady-state schedule of tasks based on the current TBO value of 300 clock units. The bars are shaded (with colors or patterns) according to the relative data set numbers shown above the bars. The TGP window has the same measurements and viewing features as the SGP window, including the time cursors. The time cursors are positioned at the far left- and far right-hand sides to indicate the TBO interval of 300 clock units as shown in parentheses. The mouse cursor (shown as a hand) can be used within the TGP (and SGP) window to point at a bar for quick access of information as shown to the right of the TGP window in figure 19 for node B. The information window shows, among other things, that task B requires two instantiations at a TBO of 300 clock units. This is also apparent by observing that there are two overlapped bars associated with task B for relative data sets 1 and 2. The circuit-imposed zero slack time of task E is portrayed in figure 19 by observing that, even though there is slack between the completion of task E and the start of task F, task D requires scheduling at the same time task E completes. Note also that due to the E $\prec$ D initial token, task D will execute on a data set injected one TBO interval later than the data set produced by the completion of task E.

Figure 20 shows how processor requirements and utilization can be shown graphically with a resource envelope diagram. The Design Tool provides a resource envelope window for both the SGP and TGP displays referred to as the "single resource envelope" (SRE) and "total resource envelope" (TRE), respectively. The TRE window for the TGP of figure 19 is shown in figure 20. Processor utilization for any time interval defined between the left and right time cursors is automatically calculated and displayed in a separate window. The processor utilization for the entire TBO interval of 300 clock units is shown in figure 20, indicating that a maximum of four processors are required with 83.3 percent utilization. The Utilization window also shows that, within the same



4 Processors... 33.3 %
3 Processors... 100.0 %
2 Processors... 100.0 %
1 Processors... 100.0 %
0 Processors... 0.0 %

Computing Effort = 1000

Total Utilization 83.3 %

Figure 20. Total resource envelope window.

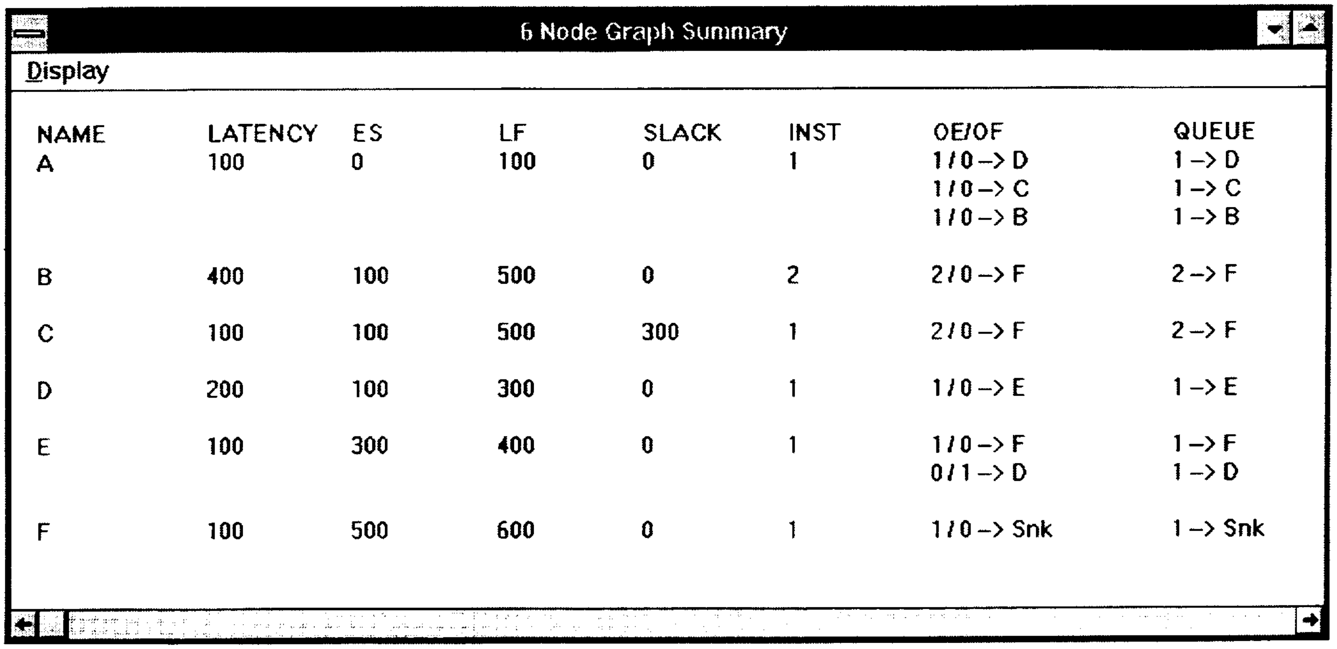
time interval, three out of the four processors are utilized 100 percent of the time and all four processors are utilized 33.3 percent of the time. The `Computing Effort` is the area under the envelope curve and is equal to TCE.

A summary of the task system $(\mathcal{T}, \prec, \mathcal{L}, \mathcal{M}_0)$ is given by a window referred to as the "graph summary window" shown in figure 21 for the four-processor, 300-clock-unit TBO performance level. The graph summary window displays the values of $\mathcal{L}$, ES, LF, slack, and instantiations (INST) for each task in $\mathcal{T}$ along with the initial tokens and queue sizes for each edge in $\prec$. The ES times shown in figure 21 are associated with the task start times in figure 18. It is apparent from this window that task C is the only task with slack (measured to be 300 clock units) as already indicated by figure 18. The graph summary window also indicates the earlier observation that task B requires two instantiations. The OE/OF column provides the initial state of the detailed Petri net model of figure 10 indicating the initial state $\mathcal{M}_0$ and maximum queue size, also shown in the QUEUE column. The QUEUE column shows that two buffers are required for the data associated with edges B $\prec$ F and C $\prec$ F.

5.1. Design Tool Use in Graph Optimization

As discussed in the previous section, the example DFG has the potential of having a speedup performance of 3 with three processors as indicated by figure 15. However, the precedence relationship $\prec$ given by the dataflow may not lend itself to this analysis in terms of requiring three processors at a TBO of 334 clock units. Note that the optimum TBO for three processors is 333 1/3 clock units. The Design Tool maintains the defined precision by rounding fractional times up to the next integer value. The graph source will ultimately be controlled to inject data at a rate 1/TBO determined by the Design Tool such that predictable performance can be attained and resource saturation avoided. The clock resolution used in the actual multiprocessing system is assumed to be the same as that defined for the tool, and therefore fractional times are rounded to the next clock unit for proper input-injection control.

The inclusion of additional precedence constraints in the form of control edges may reduce the processor requirements of a DFG for a desired level of performance. Since such a problem of finding this optimum solution is NP-complete and requires an exhaustive search, the Design Tool was developed to aid the user in finding appropriate control edges when needed and to make trade-offs when the optimum solution cannot be found or does not exist (ref. 9). The design of a solution for a particular TBO, TBIO, and R is ultimately application dependent. That is, one application may dictate that suboptimal graph latency ($\text{TBIO} > \text{TBIO}_{lb}$) may be traded for maximum throughput ($1/\text{TBO}_{lb}$) while another application may dictate the opposite. An application may also specify a control/signal processing sampling period (TBO) and the time lag between graph input $g(t)$ and graph output $g(t - \text{TBIO})$ that is greater than the lower

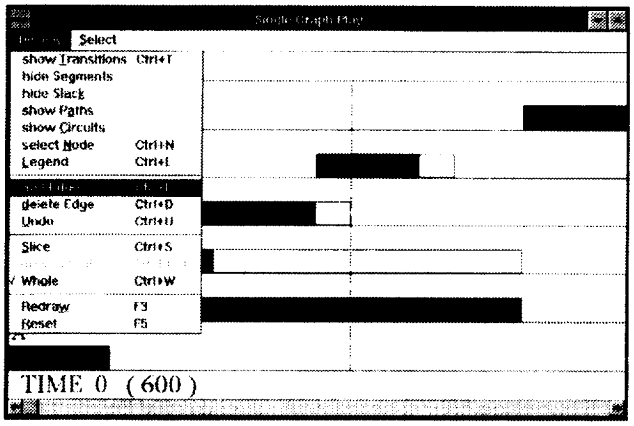


NAME	LATENCY	ES	LF	SLACK	INST	OE/OF	QUEUE
A	100	0	100	0	1	1 / 0 –> D 1 / 0 –> C 1 / 0 –> B	1 –> D 1 –> C 1 –> B
B	400	100	500	0	2	2 / 0 –> F	2 –> F
C	100	100	500	300	1	2 / 0 –> F	2 –> F
D	200	100	300	0	1	1 / 0 –> E	1 –> E
E	100	300	400	0	1	1 / 0 –> F 0 / 1 –> D	1 –> F 1 –> D
F	100	500	600	0	1	1 / 0 –> Snk	1 –> Snk

Figure 21. Graph summary window of four-processor schedule shown in figure 19 for TBO = 300 clock units and TBIO = 600 clock units.

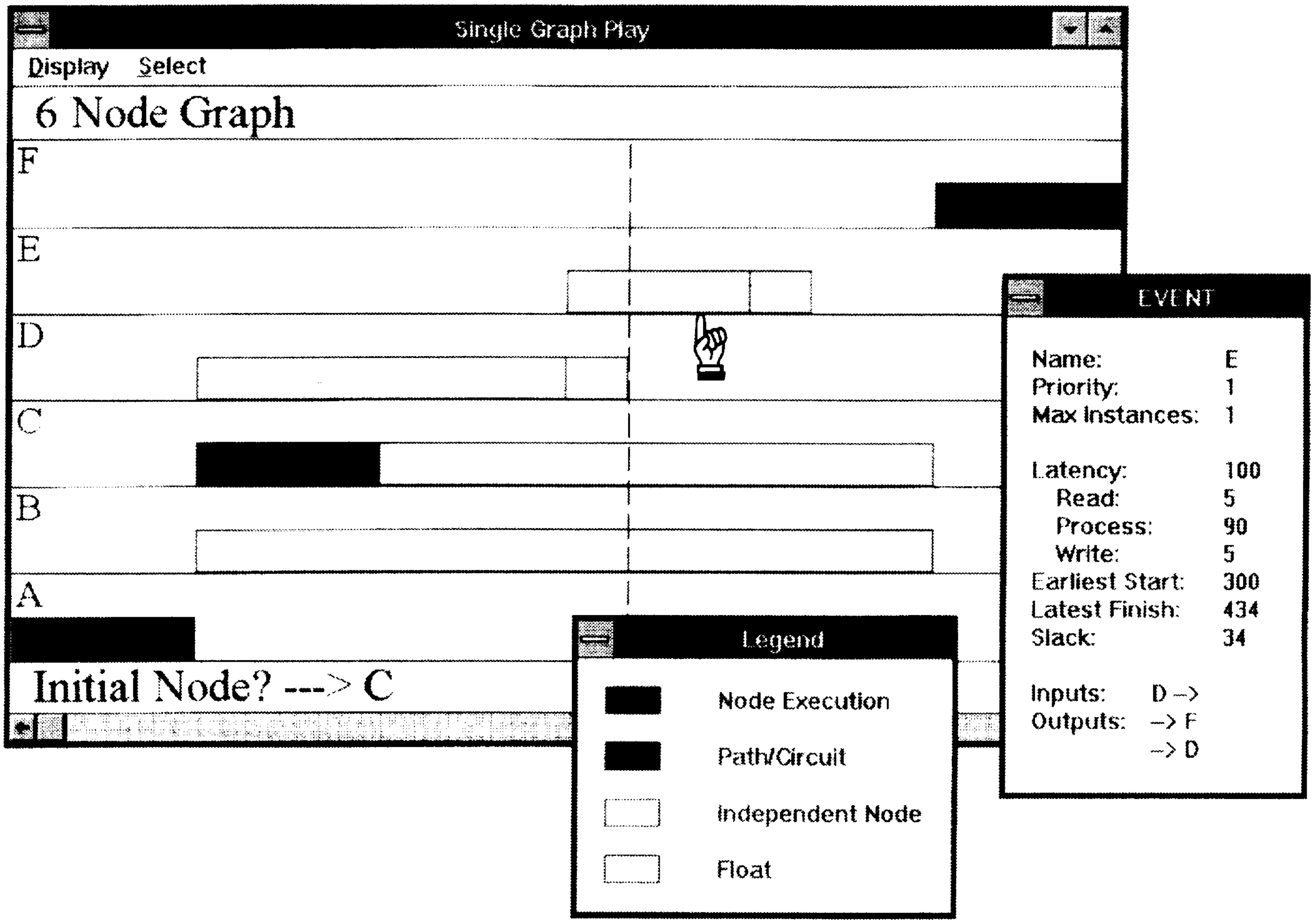
bounds determined from graph analysis, possibly making it easier to find a scheduling solution.

Use of the Design Tool for solving the optimum three-processor solution is presented as an example since the results can be compared with the theoretical results in the previous section. First, the control edge E ≺ C which eliminates the needless parallelism for a single iteration can be added from the SGP window by selecting the `add Edge` menu option as shown in figure 22. Any control edge added within the SGP window will never be initialized with tokens resulting in only intra-iteration precedence relationships. This is the desired effect with the E ≺ C relationship. Upon selecting the `add Edge` menu option, the SGP window will prompt the user for a terminal node to be delayed by the control edge. Once the terminal node (task) has been selected as shown in figure 23, all nodes (tasks) independent of the terminal node (task C) will be highlighted. These highlighted nodes become the only candidates for selection as the initial node. Selection of a dependent node is prohibited because a circuit would be generated without any tokens; this is a nonexecutable situation. The use of the information window and time cursors may prove useful in making use of slack time or delaying tasks such that any



TIME 0 (600)

Figure 22. Adding a control edge by using SGP window.



Initial Node? ---> C

Name:	E
Priority:	1
Max Instances:	1
Latency:	100
Read:	5
Process:	90
Write:	5
Earliest Start:	300
Latest Finish:	434
Slack:	34
Inputs:	D ->
Outputs:	-> F -> D

Figure 23. Selecting the initial node of control edge.

increase in TBIO is minimized. Since task C, duration of 100 clock units, has 300 clock units of slack time and task E finishes 100 clock units short of the start of task F, one can easily see that task C can be delayed behind task E without increasing TBIO. Selection of node E causes the Design Tool to create the control edge $E \prec C$, reapply the analysis algorithms, and create the expected SGP shown in figure 24.

The new periodic schedule as a result of the new $E \prec C$ control edge is shown in figure 25(a) with the processor utilization portrayed in the TRE window of figure 25(b). At this point, a search for additional precedence relationships is necessary that could effectively move the computing effort requiring four processors to fill in the underutilized idle time requiring only two processors. As noted in section 4.4, a control edge creating the precedence relationship $B \prec D$ provides a solution. Addition of this control edge is done in the same way as within the SGP window. However, unlike control edges added within the SGP window, control edges added from the TGP window are automatically initialized with tokens as required to assure the desired steady-state schedule (using the `OE` and `OF Rules`). Insertion of the $B \prec D$ control edge from within the TGP window results in the schedule and processor utilization as portrayed in figures 26(a) and (b), respectively. It is apparent from figure 26(a) with the two additional precedence relationships, $E \prec C$ and $B \prec D$, that an optimum solution for three processors in terms of throughput has been found. Note that 0.6 percent of idle time is contributed to the rounding up of the ideal 333 1/3 clock units TBO to 334 clock units for implementation purposes. As mentioned, this solution is only optimal in terms of throughput due to the 66 clock units delay of node D (indicated by the left and right cursors in fig. 26(a)). Since node D lies in the critical path, this delay results in a TBIO of 666 clock units, as shown by the LF time of task F in figure 27. The graph summary window in figure 27 also displays the control edges added for optimization, indicated by asterisks. Referring to the $B \prec D$ control edge, the `OF` equal to 1, representing the presence of one initial token, characterizes the inter-iteration relationship that is required between B and D (one TBO delay) to assure the desired schedule in figure 26(a), as expected from the analysis in the previous section.

5.2. Case Study

Another example is given in this section for the purposes of demonstrating the dependence that steady-state behavior has on $\prec$, $\mathcal{M}_0$, and TBO. The same six-node graph is utilized except for a different initial marking $\mathcal{M}_0$ and the additional precedence constraint between nodes C and B as shown in figure 28. These differences result in a new graph which is referred to as "DFG2."

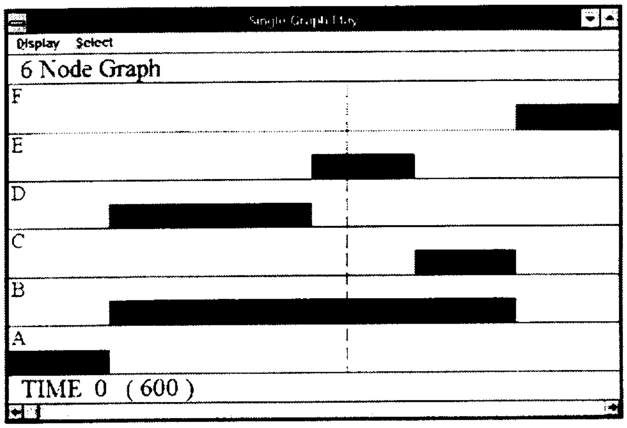
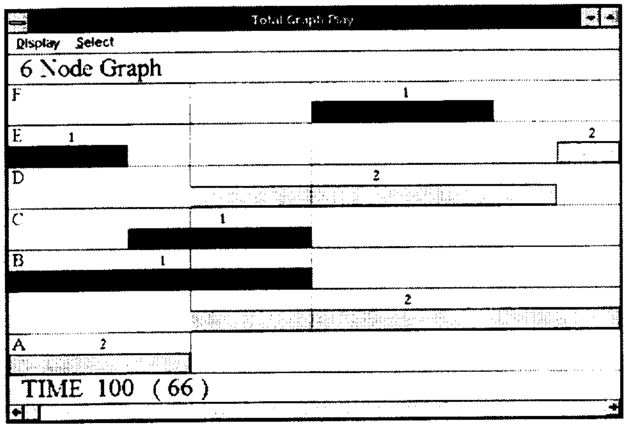
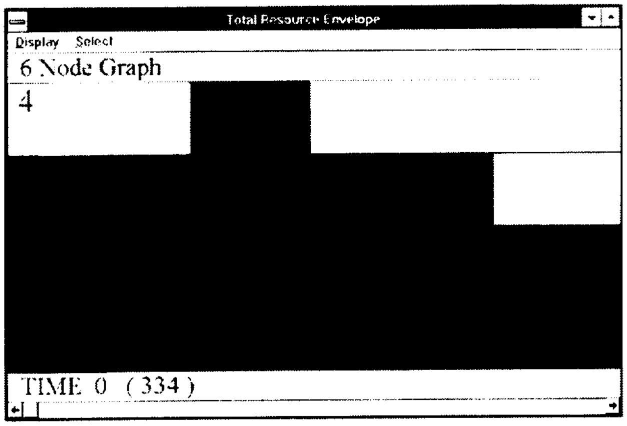


Figure 24. SGP window with control edge $E \prec C$.

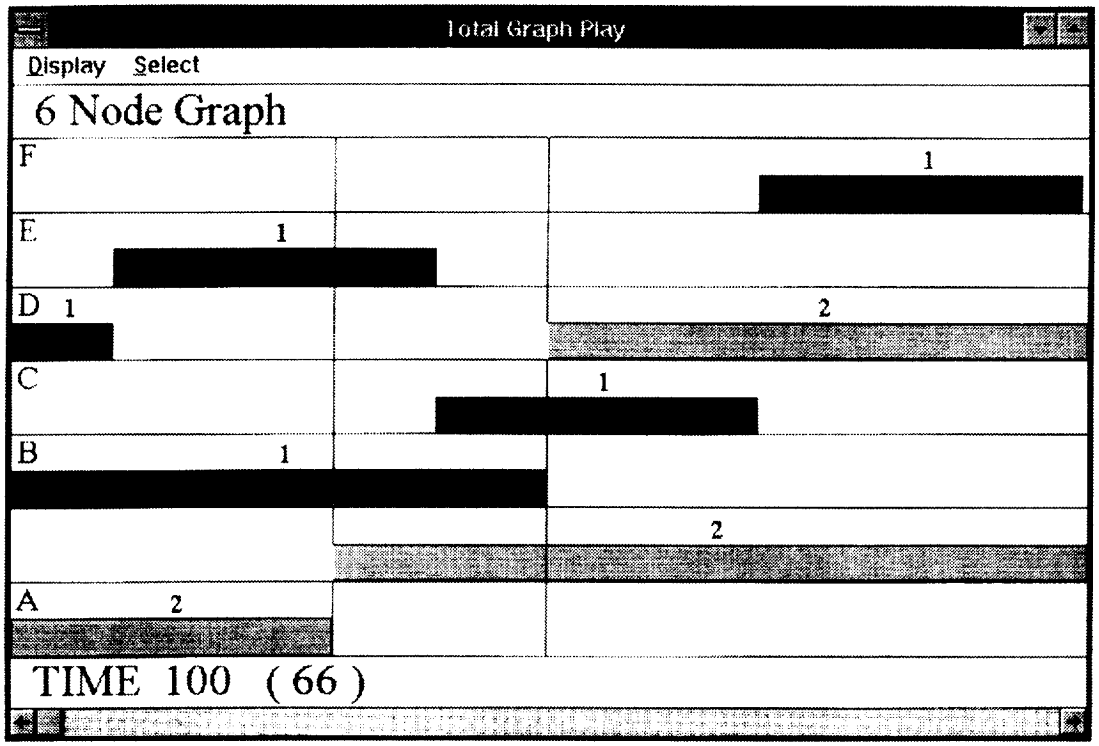


(a) TGP window.

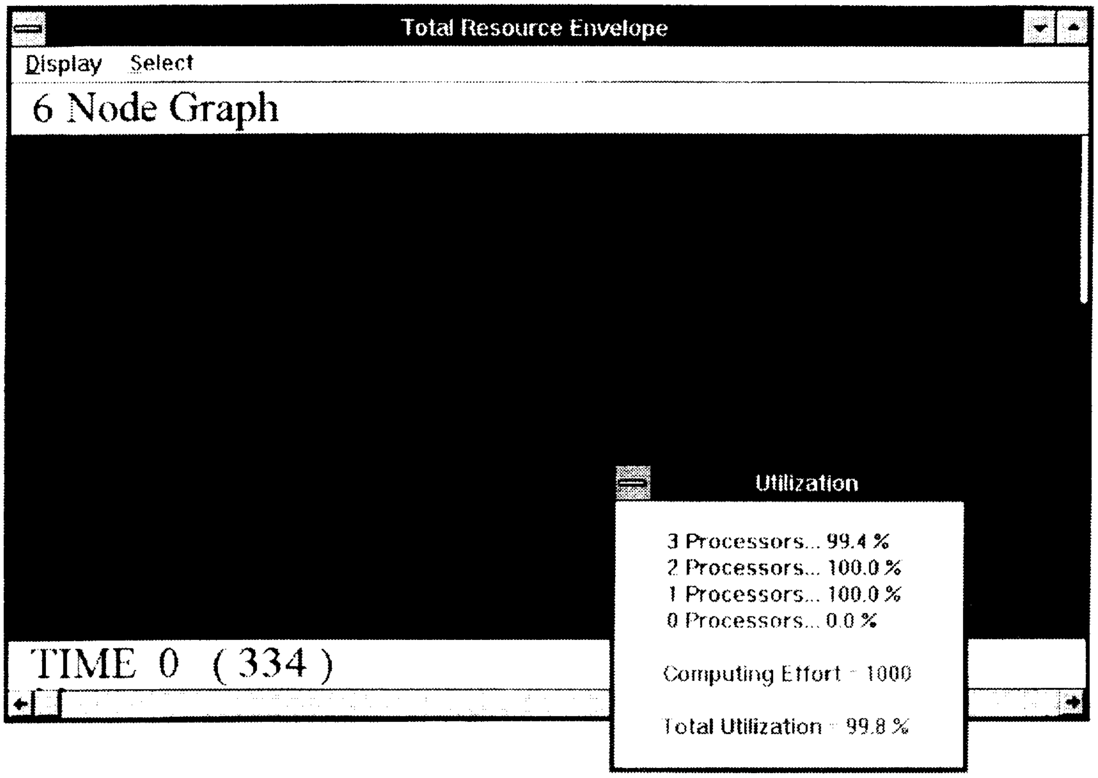


(b) TRE window.

Figure 25. Windows with control edge $E \prec C$.



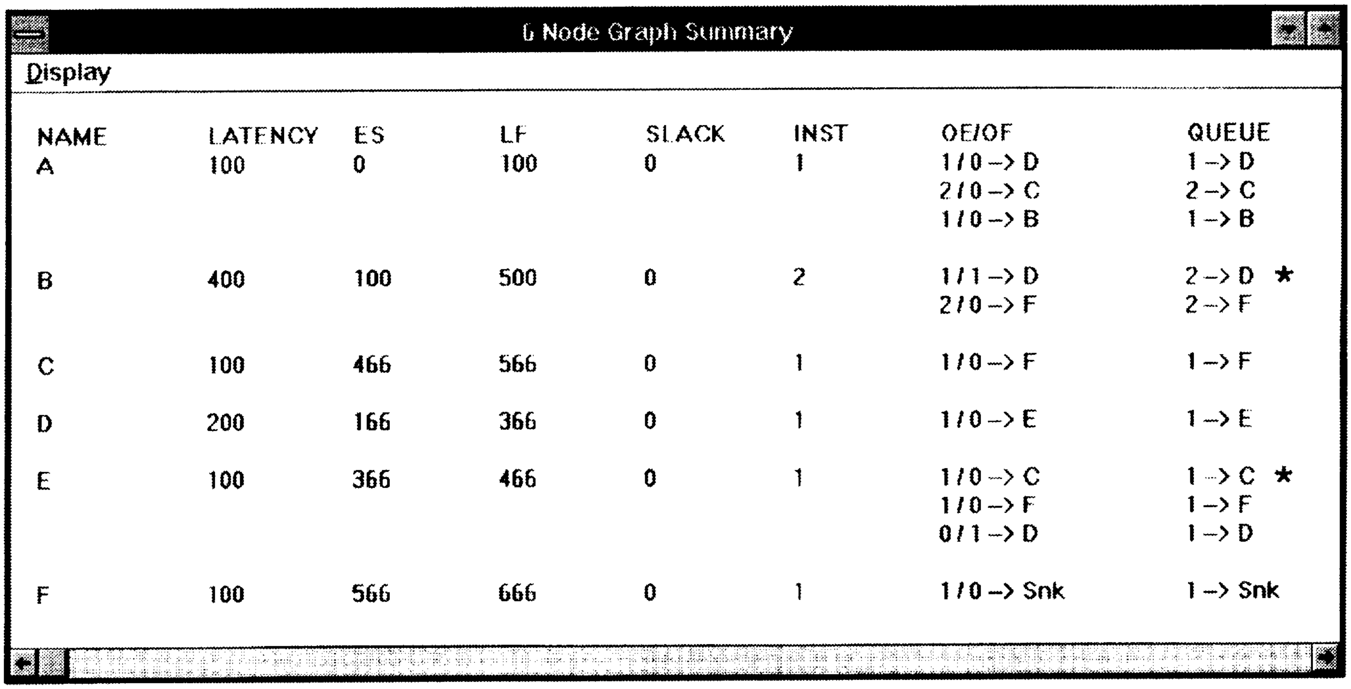
(a) TGP window.



```
3 Processors... 99.4 %
2 Processors... 100.0 %
1 Processors... 100.0 %
0 Processors... 0.0 %

Computing Effort = 1000

Total Utilization = 99.8 %
```

(b) TRE window.

Figure 26. Windows with control edges E $\prec$ C and B $\prec$ D.



NAME	LATENCY	ES	LF	SLACK	INST	OE/OF	QUEUE
A	100	0	100	0	1	1 / 0 –> D 2 / 0 –> C 1 / 0 –> B	1 –> D 2 –> C 1 –> B
B	400	100	500	0	2	1 / 1 –> D 2 / 0 –> F	2 –> D ★ 2 –> F
C	100	466	566	0	1	1 / 0 –> F	1 –> F
D	200	166	366	0	1	1 / 0 –> E	1 –> E
E	100	366	466	0	1	1 / 0 –> C 1 / 0 –> F 0 / 1 –> D	1 –> C ★ 1 –> F 1 –> D
F	100	566	666	0	1	1 / 0 –> Snk	1 –> Snk

Figure 27. Optimized graph summary window of three-processor schedule shown in figure 26(a) for TBO = 334 clock units and TBIO = 666 clock units.

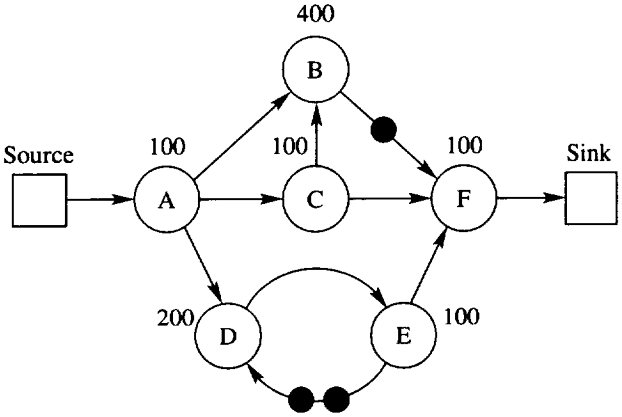


Figure 28. DFG2 with initial token on forward-directed edge.

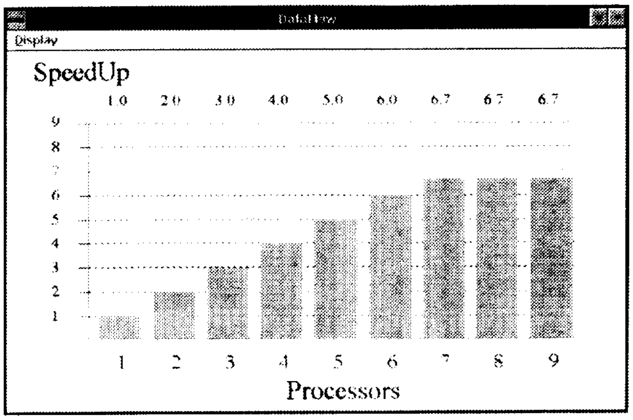
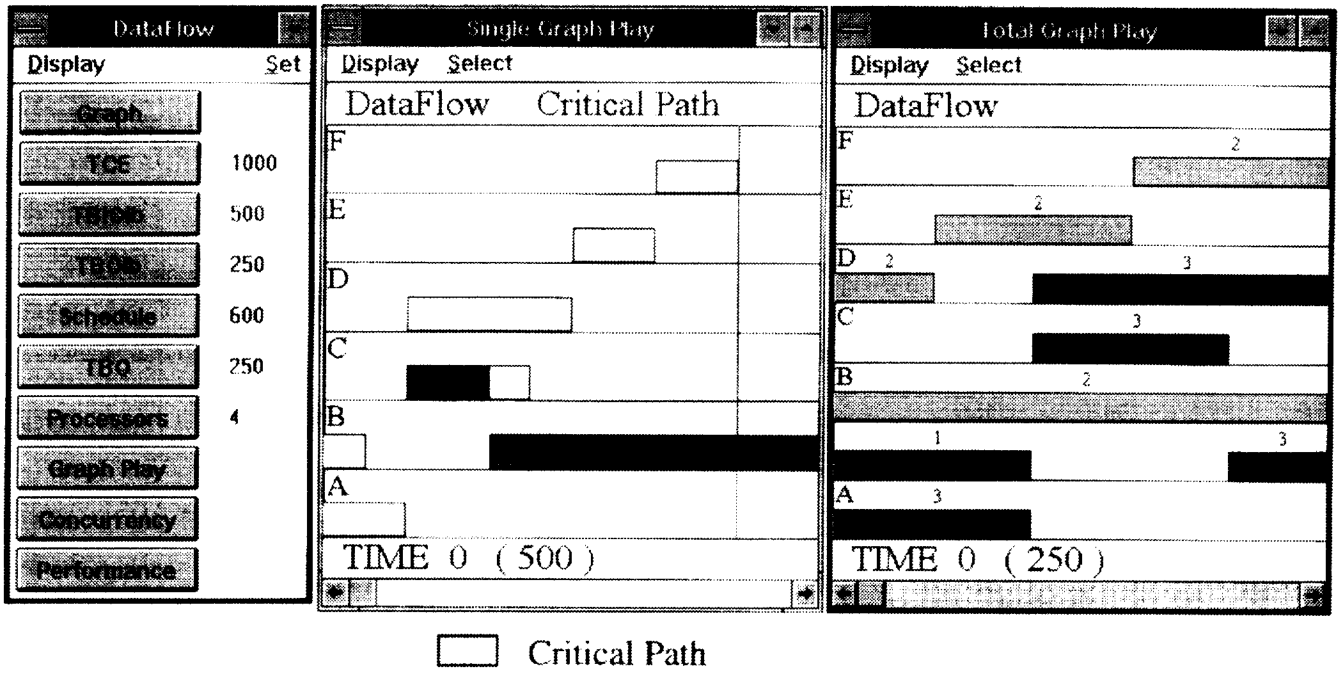


Figure 29. Speedup potential of figure 28 DFG.

As a result of the additional token in the D ≺ E circuit, the graph-theoretic speedup bound has increased; therefore a speedup capability up to seven processors (fig. 29) is provided. The initial token on the B ≺ F edge affects the steady-state performance differently by making TBIO and ω dependent on the iteration period, TBO. For the purposes of illustrating this effect, the scheduling solutions for two different iteration periods are shown. The first example shown in figure 30, which requires four processors for a TBO of 250 clock units, results in a TBIO of 500 clock units (indicated in parentheses using the SGP window cursors) which is less than the graph schedule length of 600 clock units (indicated next to the `Schedule` button). At this iteration period, both tasks B and C have slack time. The slack time of task B is shown to the left for the convenience of displaying an interval equal to the schedule time and because any delay in the completion of task B affects the execution (start time of task F) for the next data packet iteration.

The initial token on the B ≺ F edge also has the potential of causing a transient condition such that



TCE 1000

TBIOlb 500

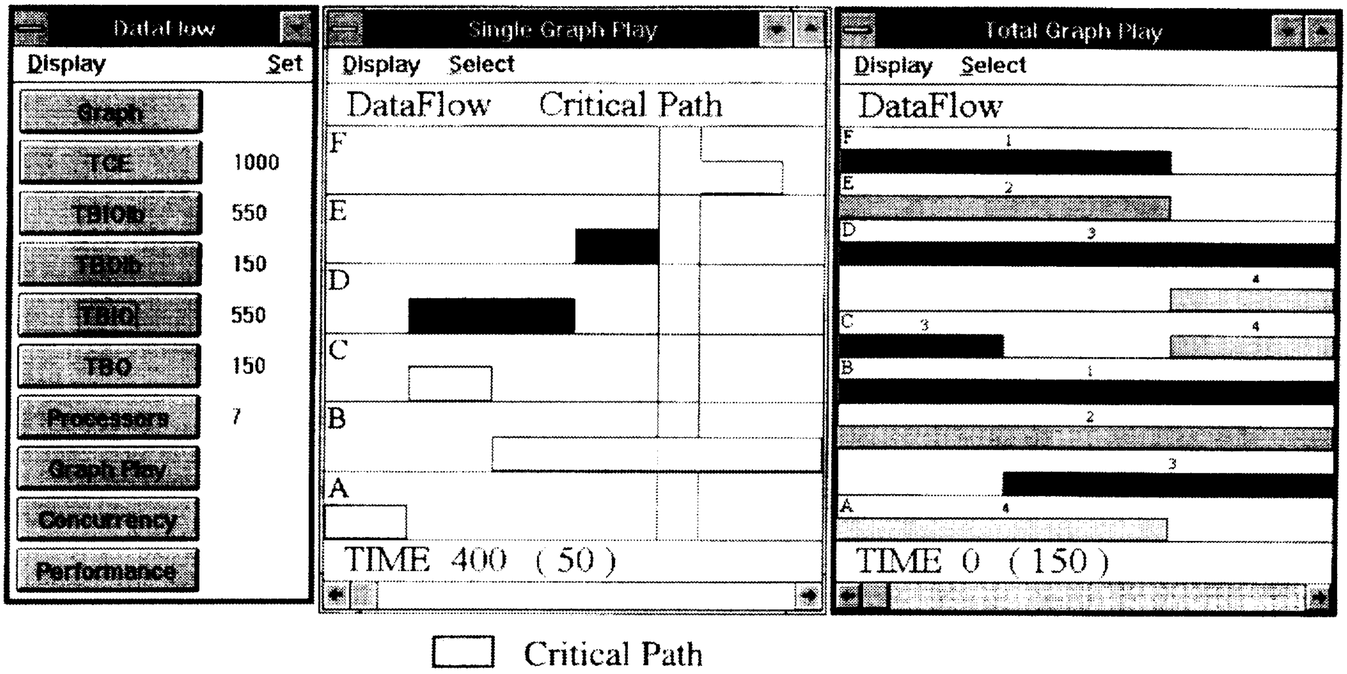
TBOlb 250

Schedule 600

TBO 250

Processors 4

Figure 30. Dataflow schedule of figure 28 for four processors.



TCE 1000

TBIOlb 550

TBOlb 150

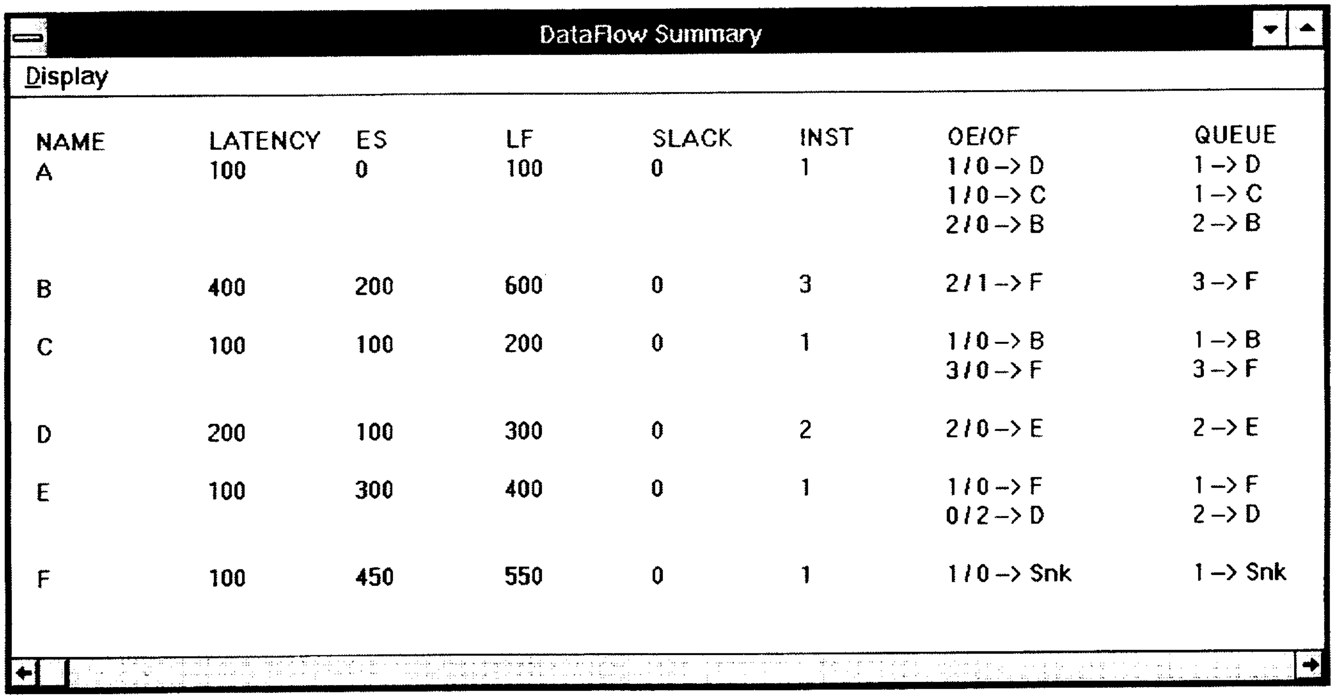
TBIO 550

TBO 150

Processors 7

Figure 31. Dataflow schedule of figure 28 for seven processors.

$SGP_{s-s} \neq SGP_{t-s}$, which has an effect on the steady-state performance. The second example, shown in figure 31 for the smallest possible iteration period of 150 clock units for seven processors, results in a schedule length equal to 600 clock units, which is still greater than the TBIO of the graph; however, the critical path has changed from the previous example. The Design Tool has found the critical path to be $A \prec C \prec B \prec F$. Also, the initial token at this TBO performance has caused task F to delay 50 clock units



NAME	LATENCY	ES	LF	SLACK	INST	OE/OF	QUEUE
A	100	0	100	0	1	1/0 -> D 1/0 -> C 2/0 -> B	1 -> D 1 -> C 2 -> B
B	400	200	600	0	3	2/1 -> F	3 -> F
C	100	100	200	0	1	1/0 -> B 3/0 -> F	1 -> B 3 -> F
D	200	100	300	0	2	2/0 -> E	2 -> E
E	100	300	400	0	1	1/0 -> F 0/2 -> D	1 -> F 2 -> D
F	100	450	550	0	1	1/0 -> Snk	1 -> Snk

Figure 32. Graph summary of figure 28 for seven processors.

(indicated by the SGP window cursors), as compared with the case shown in figure 30, resulting in a TBIO equal to 550 clock units. Because the calculated processors (eq. (10)) are equal to the seven "sufficient" number of processors (derived from the TGP window) for the optimum iteration period of 150 clock units, the steady-state schedule shown in the TGP window is an optimum solution for this example task system. The TGP window also shows that the additional pipeline concurrency allows the simultaneous execution of four data packets within a TBO interval.

Figure 32 shows the task system $(\mathcal{T}, \prec, \mathcal{L}, \mathcal{M}_0)$ summary for a TBO of 150 clock units. The LF of task F with no slack indicates that the TBIO is 550 clock units. Also, tasks B and D require three and two instantiations, respectively. As one might have expected, the queue size (memory requirements) has increased from the lower speedup example examined in the previous section (figs. 20 and 28).

5.3. Algorithm Implementation Performance

The `ES Algorithm` and the `LF Algorithm` can be executed in polynomial time. For typical graphs, the actual bound is somewhere between $O(N^2)$ and $O(N^3)$ where equation (1) provides a conservative graph-dependent bound. The C++ program code for the `ES Algorithm` and the `LF Algorithm` is included in the appendix. This section provides some performance data on the execution of these algorithms within the Design Tool.

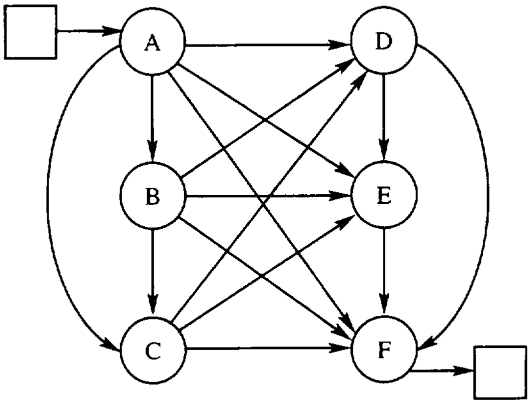


Figure 33. Test graph.

The performance results of the `ES Algorithm` and `LF Algorithm` within the Design Tool were obtained for the graphs in figures 1, 14(d), and 33. The graph in figure 33 was chosen as a good test when the graph is tightly connected. Since the three graphs have six nodes ($N = 6$) each, the worst-case complexity is given as N^3 or 216. In addition, the graph-dependent bound given by equation (1) was determined for each graph for comparison with the actual complexity. The time it takes to execute steps 1 through 6 in both the `ES Algorithm` and the `LF Algorithm` is assumed to take a constant time

Table II. Design Tool Performance Results

Graph in	ES Algorithm			LF Algorithm		
figure—	Bound	C	Duration, µs	Bound	C	Duration, µs
1	10	8	297	13	12	665
15(b)	15	10	390	20	18	920
34	64	32	934	64	32	1214

of K_1 and K_2, respectively. The actual time complexity C to complete the ES Algorithm is defined as the number of times steps 1 through 6 are executed for a given graph such that the total execution time is on the order of K_1C. Unlike equation (1) which assumes that all nodes are traversed for every path, the ES Algorithm and the LF Algorithm are more efficient in that each remembers the previous nodes and path latency covered at any given edge branch. Thus, the actual complexity C will be less than the bound of equation (1) for most cases.

The performance of the Design Tool was measured on a Gateway2000 486/33 EISA personal computer. The computer operated with a 33-MHz clock speed and contained 16 MB of RAM memory. From the performance results given in table II, the Bound (eq. (1)) and actual complexity C for the graph in figure 33 without initial tokens are equivalent for both algorithms. However, since the backward-search LF Algorithm will encounter more nodes than the forward-search ES Algorithm when virtual sinks are present, the Bound and C for the graph in figures 1 and 14(d) with initial tokens are different. Note in all cases, however, that C is less than the bound given by equation (1) indicating the degree of efficiency in the algorithms.

6. Tool Applications and Future Research

For years, digital signal processing (DSP) systems have been used to realize digital filters, compute Fourier transforms, execute data compression algorithms, and a vast amount of other compute-intensive algorithms. Today, both government and industry are finding that computational requirements, especially in real-time systems, are becoming increasingly more challenging. As a result, many users are relying on multiprocessing solutions to meet the needs of these problems. To take advantage of multiprocessor architectures, novel methods are needed to facilitate the mapping of DSP applications onto multiple processors. Consequently, the DSP market has exploded with new and innovative DSP hardware and software architectures which provide mechanisms to efficiently exploit the parallelism inherent in many DSP applications. The dataflow paradigm has also been getting considerable attention in the areas of DSP and real-time systems. The commercial products that are offered today utilize the dataflow paradigm as a graphical programming language but do not incorporate dataflow analyses in designing a multiprocessing solution. Although there are many advantages to graphical programming, the full potential of the dataflow representation is lost by not utilizing it analytically as well. In the absence of the analysis/design offered by this software tool, the commercial tool sets must rely on compile-time approximate solutions (heuristics) or run-time scheduling which often results in a trial-and-error design approach. Not only can this tool lend itself to NASA aerospace DSP problems, but it is felt that this tool has high commercial potential as well. It could be readily incorporated into existing commercial DSP tool sets to determine a desirable multiprocessing solution at compile time. Other commercial uses of this tool include scheduling of DSP algorithms for real-time applications, including those found in aircraft, automotive, and industrial processes. The tool could also provide front-end scheduling constraints for other commercial tools utilizing job-scheduling algorithms with the potential of finding better solutions.

Extensions to the Design Tool planned include incorporating heuristics to automate the selection of control edges for optimal or near-optimal scheduling solutions. Also, enhancements to the underlying model and control edge heuristics are planned which will permit the design of real-time multiprocessing applications for both hard and soft deadlines (ref. 18). For hard real-time modeling, the design would assume worst-case task latencies. It has been observed that under such assumptions, run-time behavior may result in anomalous behavior such as requiring more processors than indicated from the worst-case scenario (ref. 19). However, such anomalies can be avoided by inserting additional control edges which impose stability criteria (ref. 19). Incorporating a stability criteria algorithm similar to reference 19 would allow the Design Tool to not only determine control edges for increased performance, but to also guarantee hard deadlines. In the context of DSP systems, the Design Tool is capable of supporting only a single sampling rate per graph. Many DSP algorithms require multiple sampling rates which is equivalent to graph nodes consuming and depositing multiple tokens per firing as opposed to only one token. Enhancements are planned to the graph-analysis techniques which will support multiple sampling rates within a DSP algorithm.

7. Concluding Remarks

Graph-searching algorithms were defined and shown to effectively determine scheduling constraints on a task system represented by a dataflow graph. The dataflow graph was shown to determine performance bounds inherent in the task system, task instantiations, and buffer requirements for the data shared between tasks. Gantt charts were shown to be useful in depicting periodic task schedules, scheduling constraints, processor requirements, and processor utilization based on the dataflow graph analysis. An equivalent modified dataflow graph was presented for the modeling of initial conditions in the graph. Such initial conditions were not only shown to complicate the calculation of task mobility but may also cause a transient condition. A timing relationship imposed on the modified graph was shown to separate the steady-state behavior from the transient state. A software implementation of the design algorithms and procedures referred to as the "Design Tool" was presented and shown to facilitate the selection of a graph-theoretic multiprocessing solution. The addition of artificial data dependencies (control edges) was shown to be a viable technique for improving scheduling performance by reducing the processor requirements. The selection of an optimum solution is based on user-selected criteria, that is, a particular TBO (time between outputs), TBIO (time between input and output), and R (number of required processors) or trade-offs when a solution which optimizes all three parameters cannot be found or may not exist. Optimizations with the use of the Design Tool by inserting control edges were demonstrated.

NASA Langley Research Center
Hampton, VA 23681-0001
February 1, 1995

Appendix

Implementation of ES Algorithm and LF Algorithm

The C++ program code which implements the ES and LF Algorithms is provided in this appendix. These functions are private to the C++ Graph object which constructs and analyzes the dataflow graph. The SearchFwd function is called by the findEarliestStart function to provide a depth-first search of the graph and determine the earliest start times of all nodes. The SearchBkwd function effectively mirrors the SearchFwd function to provide a depth-first search of the graph in the opposite direction. The SearchFwd and SearchBkwd functions are used by the findLatestFinish function to determine the latest finish times of all nodes.

```
//Declaration of node and edge types
// DATA......data edges found in graph text file,
// CONTROL...control edges already present in graph text file,
// NEW.......control edges added by this tool,
// VIRTUAL...fictitious edges added to model inter-iteration dependencies, and
// SPECIAL...control edges added to source input for input injection control.

enum nodetype { NODE, SOURCE, SINK, VIRTUAL_SOURCE, VIRTUAL_SINK };

enum edgetype { DATA, CONTROL, NEW, VIRTUAL, SPECIAL };

typedef int ClockTicks;

struct Times { ClockTicksread,          //time to read input data
                         process,       //time to process data
                         write,         //time to write output data
                         earliest_start,//earliest possible start time
                         latest_finish, //latest finish time
                         fire; };       //time to fire node

class Node { char name[SIZE];  //node name
             nodetype type;    //node type
             int number,       //node #
                 graph,        //graph #
                 priority,     //task priority
                 instances,    //required instantiations
                 data_set;     //relative data set #
                 Times time;   //node times

             public:
             class Node *previous, *next;
             class Edge *input, *output;

             public/private methods...; };

class Edge { int number,       //edge #
             token_limit,      //queue size = initially empty + initially full
             tokens,           //initial tokens = initially full queue slots
             edgetype type;    //edge type

             public:
             class Edge *previous, *next;
             class Node *initial, *terminal;
             class Edge *next_input, *next_output;

             public/private methods...; };

// SearchFwd( Edge*, ClockTicks )
// Implements a forward search of the graph starting from an Edge until
// a sink is found. Used by findEarliestStart and findLatestFinish.

void SearchFwd( Edge *edgeptr, ClockTicks latency )
{
      while ( edgeptr != NULL )
            {
            if (edgeptr->next_output != NULL)
               SearchFwd( edgeptr->next_output, latency );

            nodeptr = edgeptr->terminal;

            // exclude SPECIAL edges, which terminate on sources
            if ( edgeptr->terminal->Type() == SOURCE )
               return;

            if ( latency > nodeptr->GetES() )
               nodeptr->SetES( latency );

            if ( nodeptr->Type() == NODE )
               latency += nodeptr->Latency();

            edgeptr = nodeptr->output;

            } //end while
      return;
}//end.


// findEarliestStart()
// Determine the earliest start times of all nodes by searching forward from
// all sources. Calls SearchFwd.

void findEarliestStart()
{
      Node *nodeptr;

      //initialize earliest start times to zero
      for ( nodeptr = first_node; nodeptr != NULL; nodeptr = nodeptr->next; )
          nodeptr->SetES( 0 );

      nodeptr = first_node;

      while (nodeptr != NULL)
          {
          //find and hold the place of a source
          while ( (nodeptr->Type() != SOURCE) &&
                  (nodeptr->Type() != VIRTUAL_SOURCE) &&
                  (nodeptr->next != NULL) )
              nodeptr = nodeptr->next;
          if ( (nodeptr->Type() == SOURCE) ||
               (nodeptr->Type() == VIRTUAL_SOURCE) )
               SearchFwd( nodeptr->output, 0 );
          nodeptr = nodeptr->next;

          }//end while
      return;
}//end.


// SearchBkwd( Edge *, ClockTicks )
// Implements a backward search of the graph from an Edge until a source is
// found. Used by findLatestFinish.

void SearchBkwd( Edge *edgeptr, ClockTicks latency )
{
      while (edgeptr != NULL)
            {
            if (edgeptr->next_input != NULL)
               SearchBkwd( edgeptr->next_input, latency );

            nodeptr = edgeptr->initial;

            //determine latest finish time
            if ( latency < nodeptr->GetLF() )
               nodeptr->SetLF( latency );

            if ( nodeptr->Type() == NODE )
               latency -= nodeptr->Latency();


            if ( (nodeptr->Type() == SOURCE) ||
                (nodeptr->Type() == VIRTUAL_SOURCE) )
               return;

            edgeptr = nodeptr->input;

            }// end while
      return;
}// end.


// findLatestFinish()
// Determine the latest finish times of all nodes by searching backward from
// all sinks. For sinks created from edges with initial tokens, the latest
// finish rule states: LF(Sink) = ES(Nt) + d * TBO where Nt is the terminal
// node of original edge (sink now points to this node) and d is the number of
// initial tokens on the original edge. Calls SearchBkwd and SearchFwd.

void findLatestFinish()
{
      ClockTicks ES, LF, delta;
      struct Node *nodeptr, *succ_node;
      BOOL Done = FALSE;
```

```
while ( !Done )
  {
  Done = TRUE;

  //initialize latest finish times to maximum storage value
  for ( nodeptr = first_node; nodeptr != NULL; nodeptr = nodeptr->next; )
      nodeptr->SetLF( 0x7FFF );

nodeptr = first_node;

while ( nodeptr != NULL )
  {
  //find and hold the place of a sink
  while ( (nodeptr->Type() != SINK) &&
          (nodeptr->Type() != VIRTUAL_SINK) &&
          (nodeptr->next != NULL) )
      nodeptr = nodeptr->next;

  if ( (nodeptr->Type() == SINK) ||
       (nodeptr->Type() == VIRTUAL_SINK) )

    {
    //if sink is a result of initial tokens on an edge then
    // LF(sink) = ES(terminal node) + d*TBO
    if ( nodeptr->Type() == VIRTUAL_SINK )
       {
       // node receiving tokens from sink
       succ_node = getNode( nodeptr->Name() );

       LF = succ_node->GetES() + (nodeptr->input->Tokens() * TBO);

       // If delta = EF - LF > 0 then a timing violation has been
       // detected. Must increase ES(terminal node) by delta to satisfy
       // timing relationship. After doing so, propagate the updated
       // ES time to all descendents. Note: EF of initial node is
       // equal to ES of sink.

       if ( (delta = nodeptr->GetES() - LF) > 0 )
          {
          Done = FALSE;

          ES = succ_node->GetES() + delta;

          //Delay the start time of node
          succ_node->SetES( ES );

          // Propagate the updated ES to all descendents
          SearchFwd( succ_node->output, ES + succ_node->Latency() );

          LF += delta;

          }//end if delta > 0

        }//end if virtual sink due to initial tokens
```

```
            else LF = nodeptr->GetES();

            SearchBkwd( nodeptr->input, LF );

            }//end if sink

         nodeptr = nodeptr->next;

         }//end while more paths
        }//end while not Done
      return;
}//end.
```

References

1. Deshpande, Akshay K.; and Kavi, Krishna M.: A Review of Specification and Verification Methods for Parallel Programs, Including the Dataflow Approach. *Proc. IEEE*, vol. 77, no. 12, Dec. 1989, pp. 1816–1828.

2. Culler, David E.: Resource Requirements of Dataflow Programs. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, IEEE, 1988, pp. 141–150.

3. Parhi, Keshab K.; and Messerschmitt, David G.: Static Rate-Optimal Scheduling of Iterative Data-Flow Programs Via Optimum Unfolding. *IEEE Trans. Computers*, vol. 40, no. 2, Feb. 1991, pp. 178–195.

4. Kavi, Krishna M.; Buckles, Billy P.; and Bhat, U. Narayan: Isomorphisms Between Petri Nets and Dataflow Graphs. *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 10, Oct. 1987, pp. 1127–1134.

5. Mielke, R.; Stoughton, J.; Som, S.; Obando, R.; Malekpour, M.; and Mandala, B.: *Algorithm to Architecture Mapping Model (ATAMM) Multicomputer Operating System Functional Specification*. NASA CR-4339, 1990.

6. Hayes, P. J.; Jones, R. L.; Benz, H. F.; Andrews, A. M.; and Malckpour, M. R.: Enhanced ATAMM Implementation on a GVSC Multiprocessor. *GOMAC/1992 Digest of Papers*, 181, Nov. 1992.

7. Jones, Robert L.; Stoughton, John W.; and Mielke, Roland R.: Analysis Tool for Concurrent Processing Computer Systems. *IEEE Proceedings of the Southeastcon '91*, Volume 2, 1991.

8. Storch, Matthew: *A Comparison of Multiprocessor Scheduling Methods for Iterative Data Flow Architectures*. NASA CR-189730, 1993.

9. Heemstra de Groot, Sonia M.; Gerez, Sabih H.; and Herrmann, Otto E.: Range-Chart-Guided Iterative Data-Flow Graph Scheduling. *IEEE Trans. Circuits & Syst.*, vol. 39, no. 5, May 1992, pp. 351–364.

10. Coffman, E. G., ed.: *Computer and Job-Shop Scheduling Theory*. John Wiley & Sons, Inc., 1976.

11. Som, Sukhamoy; Stoughton, John W.; and Mielke, Roland: *Strategies for Concurrent Processing of Complex Algorithms in Data Driven Architectures*. NASA CR-187450, 1990.

12. Lee, Edward Ashford: Consistency in Dataflow Graphs. *IEEE Trans. Parallel & Distrib. Syst.*, vol. 2, no. 2, Apr. 1991, pp. 223–235.

13. Som, S.; Mielke, R. R.; and Stoughton, J. W.: Effects of Resource Saturation in Real-Time Computing on Data Flow Architectures. *Twenty-Fifth Asilomer Conference on Signals, Systems & Computers*, Volume 1, IEEE, 1991.

14. Mielke, Roland R.; Stoughton, John W.; and Som, Sukhamoy: *Modeling and Optimum Time Performance for Concurrent Processing*. NASA CR-4167, 1988.

15. Murata, Tadao: Petri Nets: Properties, Analysis and Applications. *Proc. IEEE*, vol. 77, no. 4, Apr. 1989, pp. 541–580.

16. Jones, R. L.; Hayes, P. J.; Andrews, A. M.; Som, S.; Stoughton, J. W.; and Mielke, R. R.: Enhanced ATAMM for Increased Throughput Performance of Multicomputer Data Flow Architectures. *IEEE Proceeding of the NAECON 91*, Volume 1, 1991.

17. Som, S.; Obando, R.; Mielke, R. R.; and Stoughton, J. W.: ATAMM: A Computational Model for Real-Time Data Flow Architectures. *Int. J. Mini & Microcomput.*, vol. 15, no. 1, 1993, pp. 11–22.

18. Stankovic, John A.; and Ramamritham, Krithi: What is Predictability for Real-Time Systems? *Real-Time Syst.*, vol. 2, 1990, pp. 247–254.

19. Manacher, G. K.: Production and Stabilization of Real-Time Task Schedules. *J. Assoc. Comput. Mach.*, vol. 14, no. 3, July 1967, pp. 439–465.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
April 1995

3. REPORT TYPE AND DATES COVERED
Technical Paper

4. TITLE AND SUBTITLE
Design Tool for Multiprocessor Scheduling and Evaluation of Iterative Dataflow Algorithms

5. FUNDING NUMBERS
WU 233-01-03

6. AUTHOR(S)
Robert L. Jones III

7. PERFORMING ORGANZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17408

9. SPONSORIING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TP-3491

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified–Unlimited
Subject Category 61
Availability: NASA CASI (301) 621-0390

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A graph-theoretic design process and software tool is defined for selecting a multiprocessing scheduling solution for a class of computational problems. The problems of interest are those that can be described with a dataflow graph and are intended to be executed repetitively on a set of identical processors. Typical applications include signal processing and control law problems. Graph-search algorithms and analysis techniques are introduced and shown to effectively determine performance bounds, scheduling constraints, and resource requirements. The software tool applies the design process to a given problem and includes performance optimization through the inclusion of additional precedence constraints among the schedulable tasks.

14. SUBJECT TERMS
Multiprocessing; Real-time processing; Scheduling theory; Graph-theoretical model; Graph-search algorithms; Dataflow paradigm; Petri net; Performance metrics; Computer-aided design; Digital signal processing; Control law

15. NUMBER OF PAGES
40

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT